DEVELOPING A QUALIA-BASED MULTI-AGENT ARCHITECTURE FOR USE IN MALWARE DETECTION

DISSERTATION

Bobby D. Birrer, Captain, USAF

AFIT/DCS/ENG/10-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/DCS/ENG/10-01

DEVELOPING A QUALIA-BASED MULTI-AGENT ARCHITECTURE FOR USE IN MALWARE DETECTION

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

Bobby D. Birrer, BS, MS

Captain, USAF

March 2010

AFIT/DCS/ENG/10-01

DEVELOPING A QUALIA-BASED MULTI-AGENT ARCHITECTURE FOR USE IN MALWARE DETECTION

DISSERTATION

Bobby D. Birrer, BS, MS

Captain, USAF

Approved:

Richard A. Raines, PhD
Committee Chairman

9 Mar 10
Date

Rusty O. Baldwin, PhD
Committee Member

10 Mar 10
Date

Mark E. Oxley, PhD
Committee Member

9 Mar 10
Date

Steven K. Rogers, PhD
Committee Member

11 Mar 10
Date

Accepted:

Dr. M. U. Thomas
Dean, Graduate School of
Engineering and Management

16 Mar 10
Date

Abstract

Detecting network intruders and malicious software is a significant problem for network administrators and security experts. New threats are emerging at an increasing rate, and current signature and statistics-based techniques are not keeping pace. Intelligent systems that can adapt to new threats are needed to mitigate these new strains of malware as they are released. This research detects malware based on its qualia, or essence rather than its low-level implementation details. By looking for the underlying concepts that make a piece of software malicious, this research avoids the pitfalls of static solutions that focus on predefined bit sequence signatures or anomaly thresholds.

This research develops a novel, hierarchical modeling method to represent a computing system and demonstrates the representation's effectiveness by modeling the Blaster worm. Using Latent Dirichlet Allocation and Support Vector Machines abstract concepts are automatically generated that can be used in the hierarchical model for malware detection. Finally, the research outlines a novel system that uses multiple levels of individual software agents that sharing contextual relationships and information across different levels of abstraction to make decisions. This qualia-based system provides a framework for developing intelligent classification and decision-making systems for a number of application areas.

Acknowledgments

I want to thank my wife for her support during this endeavor and let her know that I love her more than anything in the world. Without her, all of this would be meaningless. I also want to express my gratitude to my committee and advisor for their guidance and patience. Their help and expertise were instrumental in making this research possible.

Bobby D. Birrer

Table of Contents

Page

List of Figures

Page

List of Tables

Page

DEVELOPING A QUALIA-BASED MULTI-AGENT ARCHITECTURE FOR USE IN MALWARE DETECTION

I Introduction

1.1 Background

Malware detectors are in danger of losing their ongoing struggle with computer viruses. In a week-long study by Symantec in November 2007, over 60% of programs being downloaded were characterized as malware [1]. This is significant since the malware being downloaded actually outnumbered legitimate programs. As this trend continues, it would appear unlikely that anti-virus companies will be able to keep signatures current to protect against all attacks, leaving systems vulnerable to exploitation. To counter the rapid development and appearance of new malware, intelligent, adaptive detection systems that are not reliant on static signatures offer a promising approach.

Adaptive systems have focused primarily on anomaly detection systems which classify programs based on their "distance" from either predetermined malicious or benign program training samples. The classification system adapts by adjusting the decision boundary used to classify programs. This technique detects some novel attacks, but it is unable to properly classify malicious programs that are very similar to benign programs and has false positives if legitimate programs deviate widely from the observed norm.

This research argues that as signature-based detection falls behind, anomaly detection systems are also unable to provide reliable protection because they suffer from unreasonable false positive rates. The models used are inherently unable to make intelligent classification

decisions because they do not consider context, intent, and end behavior. Rather, they simply look for symptoms of the malware rather than the relationships and behaviors that actually make a specific piece of code malicious. This research develops a new, robust malware detection technique that examines the system as a whole and identifies malware based on its fundamental essence or quale.

1.2 Research Goals

The goal of this research is to use qualia and hierarchical relationships to detect malware, thereby achieving engineering advantage over current techniques.

1.2.1 Examine Current Malware Techniques

To demonstrate an engineering advantage, the current state of the art must be established. Current malware and pattern recognition techniques provide a basis of comparison to determine whether any proposed technique or architecture actually improves performance. Examining the pitfalls of current techniques provides insight into what functionalities the system needs to have to defeat new strains of malware.

1.2.2 Define a General Model for the Computing Domain

Once the current state of the art is determined, the next step is to explore and define a model for the computing domain. The model needs to represent the major components of a computer system in a manner that can be exploited for malware detection. This includes splitting the domain into logical layers of abstraction and defining the major entities and their relationships between them. This information is used to drive the development of a malware detection architecture.

1.2.3 Develop a Qualia-based Architecture

One of the most important goals of this research is to develop a qualia-based architecture for malware detection. The system uses a novel representation based on hierarchical, multiple layers of abstraction and the relationships between entities to make decisions. Additionally, the system takes into account time and changes in the environment to further increase the detection ability. Ideally, the system is defined in a generic manner so it can be generalized beyond the malware detection domain. Such a system is then applicable to other domain areas and provides a way of performing more intelligent pattern recognition and analysis.

1.2.4 Discuss Possible Pitfalls and Future Enhancements

The final goal of the research is to examine the developed models and architectures to determine what obstacles remain to be overcome. After determining the possible problems, mitigating solutions must be discussed to provide a basis for future research. Additionally, avenues for further improvements should be outlined, demonstrating that the research can continue to advance the body of knowledge.

1.3 Document Overview

Chapter II discusses the state of the art in malware detection and pattern recognition. It also introduces the QUEST initiative and describes qualia and how they can create a more effective decision-making and object identification and classification systems. Chapter III outlines the requirements of a qualia-based architecture and provides a malware detection proof of concept. Chapter IV discusses the development of a novel qualia-based agent architecture and its benefits over current techniques. Chapter V describes a conceptual pattern-based representation that models the world through the use of QUEST tenets. It also discusses how this representation benefits the agent architecture. Chapter VI outlines the contributions and

future avenues of this research and concludes by evaluating how well the research met its stated goals in Section 1.2.

II Literature Review

2.1 Overview

This chapter covers the background research and knowledge of this research. It examines malware detection techniques, artificial intelligence techniques, and the concepts of qualia and describes their advantages and disadvantages. It also describes Latent Dirichlet Allocation, a topic extraction process discussed in Chapter III.

2.2 Current Malware Detection Techniques

Current malware detection and identification is reactive. Signature-based detection, the most common form of protection, requires the generation of a unique signature for each attack. With the increased number of viruses and worms being released daily, it is becoming more difficult to generate signatures quickly enough to keep up with new attacks. The following section describes current protection techniques and their strengths and weaknesses.

2.2.1 Signature-based

The effectiveness of signature-based detectors depends upon the prior effectiveness of techniques that detect, analyze, and identify malicious software. By this, signature-based detection systems match what is currently being seen with what has previously been observed. These detectors look for specific byte sequences that have been identified as being present in individual strains of malware. If the byte-sequence is not found in the program, then the program is considered to be benign. However, this absence of known byte-sequences is not sufficient to guarantee a program is safe. Many malware strains are capable of transforming and changing themselves, altering the target signature. Further, whenever new malware is released, it takes time to develop signatures and distribute them. This leaves a window of time during

which the malware remains undetected. Signature-based systems are commonly used across the networking environment and include anti-virus systems such as Symantec.

2.2.2 Normalized Signatures

Signature-based techniques often fail to detect malware that has modified functionality or transformed with a metamorphic engine. These engines use a variety of techniques including instruction substitution, instruction reordering, packing, and insertion of junk bytes [2]. One way to counter these obfuscations is by reducing the code back to a basic, normalized form. Christodorescu et al. evaluates code normalization as a way of mitigating code obfuscations by focusing on removing three main malware obfuscations: instruction reordering, packing, and junk insertion [3]. By removing these obfuscations, the malware is reduced to a normalized form that commercial tools are more likely to recognize and identify.

They generated one thousands variants of the Beagle.Y virus with three different degrees of obfuscation. These variants were normalized using the researchers' software before being fed into four commercial anti-virus tools. Their technique is extremely effective on the reordering and junk instruction transforms, boosting the detection rate to 100% because these transforms are relatively easy to normalize to a standard form. However, packing transforms is more difficult for the detectors, resulting in detection rates ranging from 75%-94% depending on the commercial anti-virus tested [3].

Walenstein et al. apply similar normalization techniques to metamorphic viruses that use instruction substitution [4]. This type of metamorphic viruses autonomously changes opcode sequences by replacing series of instructions with different instructions that perform the same actions. Since developing metamorphic engines is a difficult and time-consuming process, this severely limits the number of engines used. Therefore, they developed term rewriting rulesets

for well-known metamorphic engines that revert malware variants back to a normalized form and then be provided to a signature-based detector. This technique reduces multiple generations and variants of w32.Evol, a metamorphic virus that possesses a very robust transformation engine to a single normalized signature [4].

Bruschi et al. examined metamorphic viruses as a compiler optimization problem [5]. Many transforms metamorphic engines add are either junk instructions or perform simple operations in convoluted ways. By applying optimization techniques that remove these transforms, the malware is reduced to a normalized form. Their results against the MetaPHOR virus were successful, greatly reducing the difference between variants and the original archetype [5].

2.2.3 Semantics Techniques

Another technique similar to malware normalization uses semantic patterns for identification of polymorphic variants of malware. Christodorescu et al. indentifies viruses based on the semantic meaning of code sections rather than specific syntax and byte sequences [6]. Semantic templates are formed by abstracting symbolic information such as register names and memory addresses which can be changed through obfuscation or the addition/removal of functionality from the malware. These templates are created manually for each strain of malware and their usefulness is limited versus instruction replacement obfuscations [6].

2.2.4 Data Mining Techniques

Another method uses data mining on suspect programs. Schultz et al. identify three potential feature sets to classify executables: strings, DLL calls, and byte sequences [7]. Features are extracted from over 3,000 benign programs and 1,000 malicious programs and classified using a Bayesian algorithm. Their results successfully detect novel attacks, almost

doubling the detection rate of a signature-based scanner. Byte sequences proved to be a particularly useful feature as over 96% of novel executables were correctly classified with a 6% false positive rate [7].

Kolter and Maloof expand Schultz's work by analyzing different classification algorithms for the byte sequence feature set [8]. They used K-nearest neighbors (KNN), Naïve Bayes, Support Vector Machines, and other methods to classify the executables, achieving a success rate of over 99% for several methods. However, false positive rates were not reported [8].

2.2.5 Control Flow Graphs

Control flow graphs have been used to fingerprint and identify malicious software. Krugel et al. generate control flow graphs for executable code traveling through a network and compare them with graphs of known malicious software [9]. The graphs consist of nodes representing basic blocks of instructions. Basic blocks are a sequence of instructions that have no loops or branches that could potentially change the execution order of the instructions; therefore the instructions within a basic block always execute sequentially. Graph edges represent control structures such as branches and jumps. Graph coloring is applied to each node based on the type of instructions located in the basic block to provide additional granularity without over-constraining the fingerprints. These graphs are then broken into k-subgraphs for comparison to known worms and other traffic. Similar subgraphs in a large number of network flows, is an indication of a worm outbreak [9].

2.2.6 API Sequences

While control flow graphs provide fingerprints of the structure and function of a program, they are a very low-level representation and can be fooled by various instruction-level changes. To counter this weakness, some researchers characterize programs based on system call and API

sequences. APIs and system calls are the interface between processes and the operating system. Programs use these interfaces to access or modify other parts of the system. Therefore, monitoring the system calls allows a detection system to observe what the process is doing without having to follow the assembly-level implementation of the program. Many of these techniques take an anomaly detection approach to determine whether a program or system has been compromised [10-13].

Rabek et al. describe a system to detect injected, dynamically created, and obfuscated malicious code in programs [12]. Whenever a program is started, static analysis determines the system calls the program makes along with their location in the program's code. During execution, if malicious code injected into the program through a buffer overflow or revealed through a decryption routine attempts to perform a system call from a location that was not specified at the program's startup, the system generates an alert. This technique detects any code added or revealed at runtime, as long as it performs a system call at an unexpected location. This technique detects compromises in software, and does not detect malicious programs that do not attempt to hide themselves, such as Trojans. In addition, it is vulnerable to mimicry attacks where arguments of expected system calls are changed to perform malicious activities [12].

Li, Lam, and Chiueh present a more robust approach that not only uses the addresses of system calls, but also the ordering and arguments of the system calls to make decisions [11]. It generates a policy of normal system call graphs and includes any static arguments known at compile time. This requires an attack to mimic the normal ordering of system calls and prevents the attack from modifying any static arguments. In addition, the system inserts random null system calls at runtime to make mimicry attacks even more difficult. However, like the previous

technique, this system only protects against control hijacking attacks and not other programs such as Trojans and keyloggers [11].

Tokhtabayev and Skormin use non-stationary Markov models to differentiate between normal and abnormal system call behavior in a process [13]. Their research is particularly noteworthy because rather than making decisions based on one host, their system looks at the status of the entire network. If one host reports an anomaly, information about the events leading up to the anomaly are sent to a central server. Part of this information includes previous connections to other machines. The anomaly is compared to previous anomaly information stored in the central database. If a match is found, the system attempts to determine if there is connectivity between the two hosts reporting the anomaly. If a series of hosts report anomalies after interacting with each other, the system can determine that a worm outbreak is likely [13].

Christodorescu et al. demonstrate a system-call tracing technique where both malicious and benign programs are compared and specifications are automatically extracted from the malware [10]. The technique finds sequences present in the malware samples but not in the benign programs. These sequences of system calls create semantic signatures. Specifications from sixteen malware variants and a small set of benign programs were mined for testing, and by using these specifications they identified variants of the Netsky virus [10].

2.2.7 Logs and Other Measurements

In addition to API and system calls, other system-level measurements can determine if a system has been compromised. Resource usage such as CPU load and memory utilization, user login times, and other system attributes are monitored. These measurements form the basis for an anomaly detection system that generates alerts whenever they change from a predefined norm.

Qiao et al. use CPU usage, user login time, frequency of application launches, frequency of file access, I/O activities, networks statistics, memory usage patterns, data bandwidth, and connection speed as features to be monitored in an unsupervised learning anomaly detection system [14]. Shabtai et al. use similar features in their detection methodology . However, rather than relying on statistical anomaly detection, malicious "behaviors" such as high CPU usage and multiple network connection attempts while no user is logged into the system are flagged. This technique had not been tested against current malware [15].

2.3 Limitations of Current Techniques

While the techniques described above use a wide variety of measurements and classification algorithms, they fall into two main categories: signature detection and anomaly detection. Signature detection relies on specifying a priori what attributes or features indicate an attack. While classical signature detection is the most common form, normalized signatures, semantics signatures, misuse detection, and control graph matching are all forms of signature detection which are prone to false negatives when faced with novel attacks.

Anomaly detection methods do not require a priori knowledge of attacks for detection, potentially leading to fewer false negatives when faced with novel malware. However, anomaly detection methods assume that malware will behave significantly different than normal system operation, which is not always true. Malware writers craft their attacks to mimic normal actions or pace their execution to avoid detection. A truly skilled attacker can even slowly escalate their actions in an attempt to expand the system's definition of "normal behavior," allowing attacks that would have been caught initially to go undetected.

Aytug et al. discusses this type of behavior, known as strategic learning. In strategic learning, self-interested agents attempt to modify their attributes to receive a favorable

classification from a principal [16]. To counter these agents, the principal must make the decision boundary more restrictive, leaving marginally positive agents at risk to be classified negatively. This translates directly to the realm of malware, because anomaly detection systems generally have a high rate of false positives. Their definition of "normal" behavior has to be very restrictive to detect a large percentage of malware, leading to high false positive rates.

Decision boundary models fail primarily because malicious and benign programs perform the same basic actions. At the assembly level, they use the same instruction set to manipulate data, perform calculations, and execute behavior. Both types of programs use the same set of API and system calls to interact with the operating system and access resources. Even the high-level functionality can be similar. For example Microsoft Windows Update connects to a remote site, downloads code, and then makes modifications to the operating system—the same behavior that a Trojan installing a rootkit would exhibit.

It is not individual or even sequences of behaviors alone that makes a program malicious. It is the combination of behavior with the program's end effects, and the intent or purpose that causes it to be considered malware. However, current decision boundary models cannot capture this type of context, and these model limitations make the decision boundary model inadequate for performing an accurate classification.

2.4 Artificial Intelligence and Qualia

Qualia Exploitation of Sensor Technology (QUEST) is one initiative to overcome the shortfalls of current identification and recognition techniques [17]. The primary goal of QUEST is to gain an engineering advantage over current classification and prediction techniques through the use of qualia (singular quale). Qualia are the mental representations that "intelligent" life-forms use to process data from the sensed environment into a useful world model that can be

used to make decisions. These representations are not simply physical measurements, but are rather an abstraction of those measurements into meaningful information. For example, the wavelength of light is not a quale, but the concept of "redness" is. Qualia are context-specific, relative, and influenced by past experiences, knowledge, and are unique to the being that is creating them [17].

QUEST tenets describe an intelligent system that can process and use a qualia representation to provide an engineering advantage in classification, prediction, and decision-making. Any qualia-based system will adhere to these tenets and use them as part of the design process. This research focuses on six main tenets and their associated subtenets [18].

- Qualia
- Physiologically-Motivated Information Processing (PMIP)
- Learning
- Architecture
- Theory
- Driver Problems

2.4.1 Qualia

Qualia form the basis of QUEST as they are used to represent the world and make classifications, predictions, and decisions based on that representation. Qualia are inherently subjective and relative, depending on not just physical measurements, but history, current state, and context. It is possible that presenting the same measurements at different times or in different contexts can evoke different sets of qualia. For each set of input data, multiple qualia compete to determine which set of qualia best fit the observations of the world and form what QUEST refers to as a "plausible narrative". The set of potential qualia that best fit into a stable, consistent and useful world model are accepted and become the qualia which are evoked [18].

2.4.2 Physiologically-Motivated Information Processing (PMIP)

QUEST uses Physiologically Motivated Information Processing (PMIP) to demonstrate the realm of the possible. There is no known fundamental limit that prevents reasoning and intelligence observed in nature from being reproduced artificially in a machine. However, for QUEST to be successful, it only has to reproduce the functionality observed, not the exact implementation. Whatever physical implementation that allows humans and other natural entities to make intelligent decisions does not need to be actually reproduced [17] .

2.4.3 Learning

Learning develops a representation that captures the concept or quale of the class or category so later instances can be appropriately recognized. Qualia-based learning develops a representation that goes beyond memorizing measurements and can be exploited for more robust classification and recognition. This is arguably the most important QUEST tenet, given the overall goal of QUEST is to achieve an engineering advantage that allows a system to adapt and adjust as it acquires more information and experience and still remain consistent with its prior knowledge. Many of the design considerations for this system are based on learning and its sub-tenets of *entities*, *never static*, *kernel*, and *context* [18].

2.4.3.1 Entities

The world consists of entities which are instantiations of categories and classes. Entities are the “nouns” of the world and can be physical objects, events, or ideas. An identification system examines real-world entities and matches them to their appropriate classes. For malware detection research, relevant entities include benign and malicious programs, files, connections, hosts, networks, users, events, and other components of an information system.

2.4.3.2 Never Static

A major premise of QUEST is intelligent creatures in nature update their world model to adapt to new information and experiences. This allows better reactions and even anticipation of future events. This type of adaptation is especially important for malware detection, as new attacks and malicious programs are being developed at an ever-increasing rate. A system that can adapt itself to new threats with little or no human intervention would provide a significant increase of protection over current techniques.

2.4.3.3 Kernel

QUEST states that a qualia-based representation is extracted from the world by the qualia kernel, capturing the vital aspects of the environment. The kernel uses that representation to interpret the environment and reason about future actions.

2.4.3.4 Context

People often use context clues to better understand their environment and make predictions and decisions. If a person is told that the room they are about to enter is a conference room, they can use that context to predict the presence of objects such as a table, chairs, or a projector. Context plays an extremely important role in the computing realm as programs are made from a finite set of instructions. Any program ranging from Microsoft Word to the Blaster worm is written using the same basic instruction set. Combinations of executed instructions, input, and the environment in which they are executed are what differentiates programs. Even the same program could be benign on one system, but considered malicious by the administrator of another system.

For example, Netcat.exe is a program used by system administrators to perform port forwarding, remote administration, and other useful functions. However, it can also be used as a

hacker tool to provide a backdoor to a system. The program did not change, but the context in which it was used did. Ford provides a rather interesting example of such a case [19].

Dr. Ford has a program on his virus testing machine called qf.com. qf.com will format the hard drive of the machine it is executed on, and place a valid master boot record and partition table on the machine. It displays no output, requests no user input, and exists as part of the automatic configuration scripts on the machine, allowing quick and easy restoration of a "known" state of the machine. Clearly, this is not malware.

- *If I take the executable, and give it to my wife and tell her what it is, is it malware?*
- *If I take the executable, and give it to my wife and do not tell her what it is, is it malware?*
- *If I mail the executable to my wife, and tell her it is a screen saver, is it malware?*
- *If I post the executable to a newgroup, unlabelled, is it malware?*
- *If I post the executable to a newsgroup and label it as screensaver, is it malware?*

Situations such as this are why current techniques are prone to failure. There is no physical attribute or single behavior that indicates a program is malicious, but rather maliciousness of a program is determined primarily by the context it is used in. Section 4.3 defines context and discusses how it modifies the perception of an entity.

2.4.4 Architecture

QUEST proposes a basic architecture to process and form qualia. Sensors provide raw data to the system which convert this physical data into a qualia model. The model harnesses both feed forward and feedback paths to evoke, interpret, and modify qualia. The feed forward path takes the sensor data and applies it to multiple, competing paths to determine which qualia to evoke. The feedback path takes the results and uses them to modify the model and to affect future feed forward loops. The system also applies both provided knowledge such as physic models and personal experience to influence its decisions and modify its qualia [17].

2.4.5 Theory

QUEST recognizes the need for mathematical formalisms to show the fundamental limits of qualia-based computing and apply it to other domains. Having a fundamental limit such as Nyquist's Theorm provides a means of quantifying improvements in performance that are independent of data set or implementation. Additionally, QUEST recognizes that non-algorithmic approaches might be required to achieve a solution.

2.4.6 Driver Problems

QUEST attempts further understanding of qualia-based solutions through the use of application-specific driver problems. Lessons learned in each application area can be applied to the other driver problems to gain an engineering advantage in a wide variety of research disciplines. Examples of driver problems include biometrics, structural health monitoring, and malware detection, which is the focus of this research.

2.5 Latent Dirichlet Allocation (LDA)

A mathematical model being used to extract meaningful groupings and compound concepts is the Latent Dirichlet Allocation (LDA) model [20]. LDA is a generative model that explains sets of observations through unobserved groupings. It is commonly used in topic modeling of documents, where observed words are explained through mixtures of topics in a document. In the model, each document is assumed to have a Dirichlet distribution of topics which drives the distribution of words for that document. The LDA authors define words, *w*, as the basic unit of discrete data from a finite vocabulary, *V*. Each document consists of a sequence of words chosen from the vocabulary, and a grouping of *M* documents makes up the corpus, *D*.

The LDA model assumes that documents are generated by first selecting the number of words, *N*, in the document and then choosing a θ parameter that represents the distribution of

topics within that document based on the Dirichlet parameter, α. For each word in the document, *w*, a topic, *z*, is randomly chosen based on the distribution of theta is used to generate a word based on β. β is the conditional probability of a word being chosen given a specific topic and is fixed for a given corpus. [20]

For example, consider a model of a document about a pet store. The pet store "theme", represented by θ, drives the topics such as "cats", "dogs", "customers", "grooming" present in the document. Each of the topics drives the individual words that make up the document. For example, the "dogs" topic would drive words such as "ball", "fetch", and "kennel". Figure 1 shows a graphical representation of the LDA model, with circles to represent the parameters. The boxes or plates represent the repetitions of selections. The circle for the individual words selected is shaded because it is an observable parameter. The outer plate represents the repeated documents and the inner plate represents the repeated choice of words. [20]

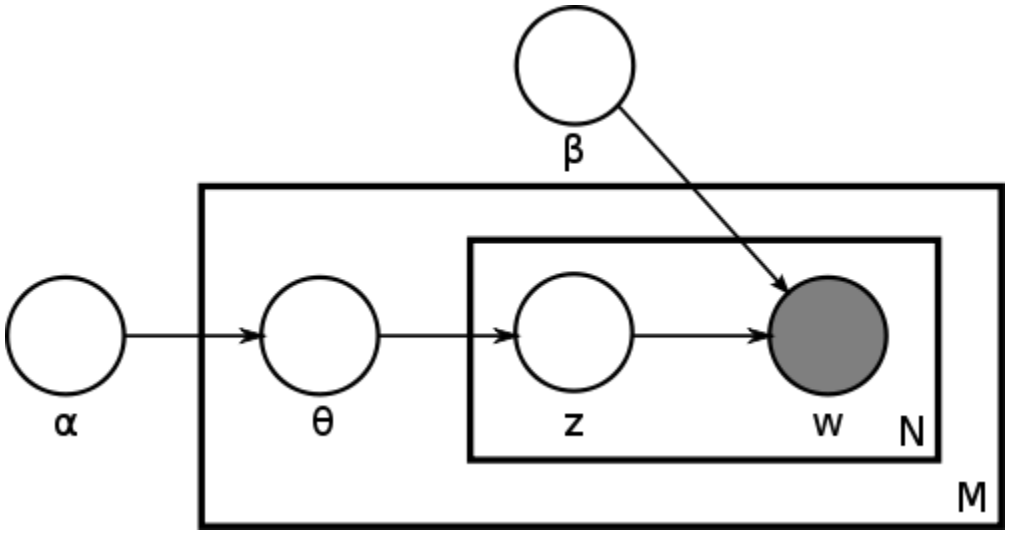

Figure 1. Plate Notation of the LDA Model [20]

Algorithms for both inference and parameter estimation are provided in the seminal LDA paper and applied and the technique is applied to an 8,000 document corpus with a vocabulary of

16,000 words. The LDA model successfully clustered words from the documents into meaningful topics that could then be used as features for document classification. Xiaogang applied LDA to video sequences of busy metropolitan scenes and clustered similar movements such as cars driving down a street and pedestrians using crosswalks[21]. Different co-occurring groups of similar movement were clustered to form interactions such as traffic blocking crosswalks. The system detected abnormal behavior such as jaywalking.

2.6 Summary

This chapter describes current malware detection techniques and the issues they face in keeping pace with novel attacks. It details the subjective nature of malware and how malicious programs adapt themselves to avoid detection. The chapter also describes the QUEST initiative's tenants and its goal to improve malware detection through the use of qualia-based detection. Finally, the chapter describes Latent Dirichlet Allocation, a mathematical technique which can be used to abstract concepts from unlabeled data.

III Methodology

3.1 Overview

This chapter describes the process of developing the requirements and general design for a qualia-based malware detection system. It outlines desired QUEST-based functionalities and how they are used to provide an engineering advantage in domains such as malware detection. The chapter then describes the design process of a basic model and kernel design and how it is applied to malware. Finally, the viability of this proof of concept implementation is demonstrated and is expanded upon in Chapter IV.

3.2 Qualia System Requirements

The goal of this research is to design a qualia-based system that makes intelligent decisions that can be applied to the multiple disciplines and application areas while achieving an engineering advantage over current techniques. QUEST uses observations from nature to determine what functionalities are possible in an artificial solution. Considering the QUEST tenets, the system should have the following basic capabilities to achieve an engineering advantage over current methods:

- Identify/classify objects based on relationships and behavior
- Use information from multiple abstraction levels
- Incorporate provided knowledge
- Learn from experience
- Make decisions based on incomplete data or assumptions
- Predict future behavior and events based on past behavior and events
- Weighs multiple predictions against each other
- Reconcile observed behavior with predicted behavior and past events
- Generate a plausible narrative of what is happening in the world

3.2.1 Identify/Classify Objects Based on Relationships and Behaviors

One of the first tasks of an intelligent entity is to identify and classify the other objects and entities in its surrounding environment. Only after the entity has established a basic understanding of its surroundings can determine what is happening in those surroundings or how to interact with them. Proper identification provides the basic foundation needed to execute other cognitive functions, and is important for the performance of the rest of the system. QUEST's context tenet states that identification and classification of individual objects is not done in isolation [17]. How the individual objects interact and are related is a vital part of determining what they are and drives the interpretation of the world. A system that can use this type of context would be capable of making better decisions than a system that only looks at the objects in isolation.

3.2.2 Use Information from Multiple Abstraction Levels

Closely related to the idea of context is an entity's ability to process information into multiple levels of abstraction. Qualia are abstractions that entities develop because they are more consistent, stable, useful, and easier to manipulate and process than the raw data [17]. Likewise, combining multiple qualia into more general abstractions provides higher level qualia that can be processed and operated on in different and possibly more efficient manners. For example, a human driver does not have to learn how to drive every make and model of car. Once a driver understands how to drive a specific car, he or she can use abstraction to apply that knowledge to all cars, only making minor adjustments for specific models. Abstraction also allows higher level assumptions and processes to affect how lower level identification and processes occur. A human walking into an office expects to see a desk, bookshelves, and chairs, and so primes their "identification system" to look for those items. Conversely, someone seeing

those items in an unknown room is likely to classify that room as an office. By comparing information at multiple layers, a system can develop a more robust and accurate model of the world which it can apply to making better decisions.

3.2.3 Incorporate Provided Knowledge

Intelligent entities start their "lives" with a basic understanding of how the world works and how to interpret, model, and interact with that world. This fundamental knowledge provides the basis for the entity's reasoning ability. An example of this fundamental knowledge in nature is instinctive or innate behaviors that are not learned, but rather passed through genetics known as genetic knowledge. In addition, intelligent entities in nature also accept knowledge from other entities. This has the effect of preventing an entity from having to relearn something a similar agent already has spent time and effort learning. This is especially useful in an artificial intelligent entity, as it would allow a user to provide knowledge to the system without having to train it through the use of repeated examples.

3.2.4 Learn from Experience

Because the world is not static, intelligent entities also have to adjust, modifying and expanding their knowledge to keep current with the changes in their environment. This updated knowledge comes through experience and repeated training. In nature, this training may or may not be supervised. Likewise, in an artificial system, the training can be supervised, unsupervised, or a combination of the two depending on the implementation and available data sets. Training can modify the entity's knowledge by changing how the entity perceives certain inputs, updating how the entity models the world, or even modifying the reasoning process. The system's ability to learn is vital, especially in application areas such as malware detection where the environment changes constantly.

3.2.5 Make Decisions Based on Incomplete Data or Assumptions

A powerful ability of humans and other intelligent entities is their capability to make assumptions and/or fill in gaps in the data they receive from their sensors. This allows them to operate in the world even if they have faulty sensors or imperfect data. For example, humans are able to identify a car even if large portions of it are hidden from view. Their minds are able to fill in the gaps that are obscured from their view, making identification still possible even without a perfect observation of the world. Being able to operate and make accurate decisions without perfect data is especially important in malware detection as almost all malware uses obfuscation in an attempt to hide portions of their code. If a malware detector can identify malicious software even if parts of the functionality still remain hidden, it would greatly limit the obfuscations' ability to protect the malware from being found and removed.

3.2.6 Predict Future Behavior and Events Based on Past Behavior and Events

While identifying entities in the world and their relationships is important, being able to predict how they will change in the future is possibly even more important. This allows the entity or system to be prepared for the changes and react to them. Knowing that brake lights on a car is an indication that the car will slow and possibly stop allows a driver following that car to slow down and avoid a potential collision. Additionally, being able to predict future events allows the system to check its current world model and knowledge. If the system makes a prediction and it comes true, then that is an indication that the system has a world model and kernel that is working properly. This is discussed further in Section 3.2.8.

3.2.7 Weigh Multiple Predictions against Each Other

The system needs to be able to compare multiple predictions against each other and determine which one makes the most sense given the current world model. These competing

predictions can have similar probabilities initially, but as more observations come in, the probabilities are adjusted to account for the new data. By examining multiple possibilities, the system can use hypotheses and assumptions to determine which prediction is more plausible. For example, if the system is unable to choose between two competing hypotheses, it can try make assumptions by filling in additional data that it did not actually observe and see how that affects the predictions.

3.2.8 Reconcile Observed Behavior with Predicted Behavior and Past Events

As mentioned in Section 3.2.6, predicting future events provides the system a way to check the validity of its model and reasoning process. If a prediction is incorrect, it could be an indication that the sensor data, world model, processing kernel, assumptions, or other portions of the system were inaccurate or faulty. If the system can go back and examine the reasons it predicted a certain outcome, it can correct the part of system that causes the incorrect prediction. This is especially useful when operating in an environment with incomplete data, as the current world model can be used to determine likely values of unobserved data at an earlier time. For example, if at T=0 only a small portion of a car is visible from behind a bush, the system may be unable to identify what is actually behind the bush. However, if at T=1 a car moves from behind the bush, that allows the system to add the data that a car was behind the bush to the previous model.

3.2.9 Generate a Plausible Narrative of What is Happening in the World

The final and arguably most difficult task of a QUEST-based system is to combine the identifications, predictions, assumptions, and other parts of the world model and kernel into what QUEST calls a "plausible narrative." A plausible narrative is the underlying or driving force on a specific part of the environment, and can be thought of almost as a summary of the events

occurring in the world. An example of a narrative is "Joe goes to work at 7:00 am." This narrative drives and explains the events and changes in the world such as Joe waking up, getting dressed, eating breakfast, and driving to work. Each of these actions is caused as part of the overall driving action of Joe going to work. Plausible narratives provide the system with a way of "understanding" what is occurring in the world and make its predictions and decisions according. It provides a way of seeing where individual events, actions, and entities fit in a larger scheme and allows the system to change how it perceives current data based on what has happened in the past or what is expected in the future.

3.3 Basic Architecture Design

From the QUEST architecture tenet, the system consists of a set of sensors, a world model, and the kernel that operates on the model. The sensors provide data that the system reads, converts, and stores in the model. The model represents the system's perception of the world in a manner that can be efficiently processed and operated on by the kernel. The kernel uses the current world model to make decisions and predictions and is responsible for updating the world model as new data is observed. Both the model and the kernel are dependent on each other, and must be designed to work together. Section 3.4 discusses the development of a model and kernel for the QUEST driver problem of malware.

3.3.1 Model Design

The model stores the system's knowledge and interpretation of the environment. It must not simply store reams of physical measurements, but instead represent information through the use of relative, subjective qualia. These qualia will be unique to that system and will change as it incorporates new information that changes its perception of the world. Further, it must store these qualia in a way that allows the kernel to efficiently access and interpret them to make

predictions about future events. Sections 3.2.1 and 3.2.2 assert that a qualia-based system must be able to model the relationships between entities and represent various levels of abstraction. Therefore, a qualia-based model must be able to capture these same relationships, making a hierarchical model a likely starting place for the design.

3.3.2 Kernel Design

While the model stores the information about the environment, the kernel is responsible for processing and interpreting the model and making decisions and predictions regarding future states and behavior. The kernel must use both learned and provided knowledge to examine not only the current state of the model; but also past states. It should use context and look at the relationships of all the objects and entities and not focus on isolated pieces of the representation. One manner of approaching this problem with a hierarchical model is to perform analysis separately at all abstraction levels and also an overall analysis of the entire model and compare the individual results. This is discussed further in Section 3.4.2.

3.4 Malware Driver Problem

The QUEST driver problem of malware detection facilitates the development of both the model and the kernel for a proof of concept system. Malware detection is one of the driver problems for QUEST because malware is a concept that is hard to define in objective terms. Because it is a subjective concept, it is difficult to capture and characterize with standard techniques. Malware is extremely context dependent, with different experts defining it differently. Even anti-virus companies cannot agree on a definite taxonomy. This is further complicated by the fact that one user can use a program on a system legitimately, while another user can use the exact same program on the exact same system for malicious purposes. If one was to ask a system administrator or an anti-virus analyst for a definition of what they looked for

in a malicious program, it is unlikely they would reply with a more concrete answer than "I know it when I see it."

A computer system capable of making that type of judgment call based on examining the entire situation and current context would provide a distinct engineering advantage in the malware domain as it would reduce the need for human generated signatures, allowing it to keep up with the release of new, novel malware.

3.4.1 Malware Model

For the system to be effective in detecting malware, a model for a computer system and network must be defined. Applying the QUEST tenets, the proposed model is made up of entities and the relationships between them. Defining entities and relationships in the cyber realm is difficult as it stretches from the user down through applications and software to the physical implementation of the system. Further, individual systems can be connected in large networks to share information and resources. Because of this complexity, the entities, relationships, and layers must be carefully considered.

The first consideration is dividing the domain into useful layers such as: physical, program, host, and network layers. These layers are chosen because they form well-defined boundaries that are meaningful to a human observer. The physical layer consists of the hardware used to implement the system including disk drives, memory, processors, and input devices. The program layer focuses on individual programs and software running on the system. The host layer consists of the relationships between programs and other components that form the file and operating systems of the host. The network layer follows the interactions of multiple hosts and their communications across the internet. While additional research may indicate the need for additional layers such as a processor layer that examines the operation of the processor such as

cache hits/misses, core load, and temperature, the layers described above provide a starting set of layers for the model.

Having established the proposed layers, the entities and relationships for each of the layers are defined. The first layer is the physical layer which is based on the Von Neumann architecture [22]. Figure 2 shows a basic diagram of the architecture [23]. This architecture consists of five main components: memory, control unit, arithmetic logic unit (ALU), input, and output. In the Von Neumann architecture, the memory contains both data and instructions which can be read or written by the control unit and ALU. Individual addresses or blocks of memory could be represented individually to allow for the differentiation between accesses of different instructions or data structures. The control unit provides the direction for the system by determining what operations to perform and when. The ALU performs the actual mathematical operations such as integer math, bitwise operations, and bit-shifting. The input and output components provide communication between the ALU and the user or storage such as the hard disk. In addition to these components, other physical devices such as fans, power supplies, and expansion cards could also be added to the model, as changes in their operation could indicate malicious activity.

The proposed relationships between the physical layer entities are basic read and write to reflect the flow of data between the components. Additional research may indicate the need for more complex relationships, but these will provide a basic functionality for initial testing and development.

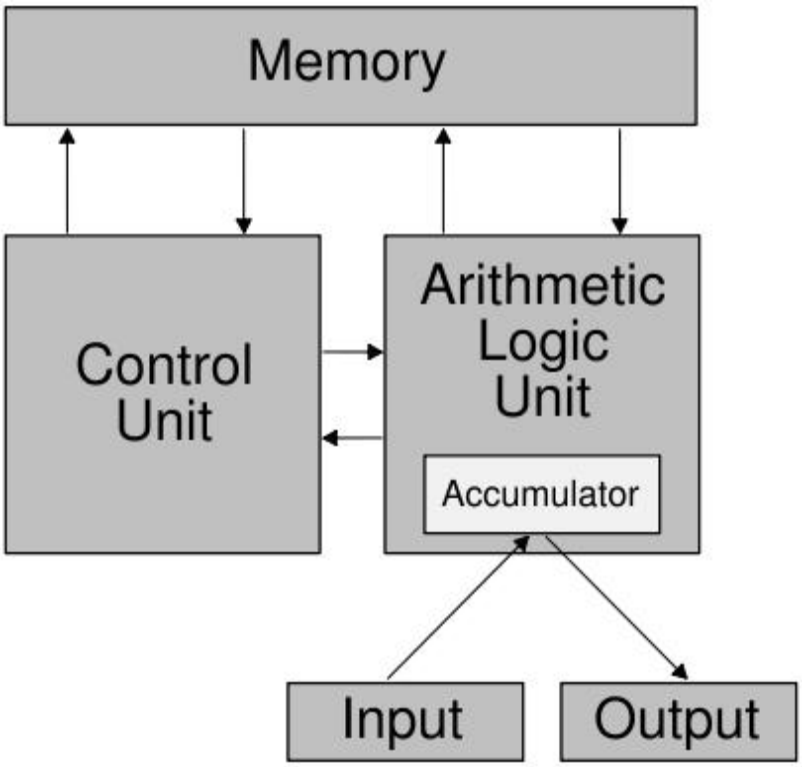

Figure 2.Von Neumann Computer [23]

The program layer consists of the individual programs on a system and the instructions used to implement them. This layer focuses on each program individually and does not follow their interactions with each other as the host layer does. The proposed entities for this layer include instruction blocks, data structures, system and API calls, and register contents. The X86 assembly language instruction set is the basis for choosing the entities and relationships as it is widely used.

Representing instructions is difficult because adequate granularity must be provided without representing each individual instruction. A control flow graph representation of programs provides a reasonable solution [9]. Each basic block of instructions is represented as a unique entity according to its basic instruction type. So the general semantics of the block is determined without having to model every instruction. Likewise, each data structure is

represented as an entity to differentiate between data accesses. Each data structure entity could also store the value of the data being held within that structure, as the value of the data affects the program's execution. However, this level of detail might be unnecessary and could be abstracted similarly to instruction blocks.

It should be noted that differentiating between instructions and data structures can be difficult in a program, as they can share the same memory space. Dissemblers can have difficulty determining beforehand the sequence of instructions to be executed, especially when the program has been obfuscated in some way. This difficulty can be mitigated through either the use of better dissemblers or by tracking the instructions as they get executed rather than trying to perform a static analysis.

System and API calls provide the interface between the program and the operating system and are invoked in a program in the same way as regular instructions. Each system call provides a basic and well-defined functionality to the program and can be represented and colored like blocks of instructions. Because system calls are themselves series of instructions, they often provide higher-level functionality that is easier to understand for a human observer.

Registers are represented similarly to data structures as they hold data between calculations and instructions use them in the same way as memory locations. The main difference is that their location and size are both fixed values. Each register is represented as its own entity to allow the system to differentiate between operations on different pieces of held data.

The relationships between the various entities are made up of basic read/write/execute links. Instructions in the X86 instruction set include stack instructions, integer ALU instructions, floating point instructions, data manipulation instructions, or program flow instructions which

can all be represented with read/write/execute links. Each instruction accesses or modifies data locations or registers and/or manages program control flow.

The combination of entities and links provides a basic outline of the program, showing what portions of the code modify data sections or make calls to the operating system. This provides a basic abstraction and representation of the program without copying it verbatim. Similar representations that focus on control flow have been shown to be effective in detecting some types of malware, so the added granularity and detail could potentially increase the detection effectiveness.

The host layer consists of the software that makes up an individual computer such as the operating system, programs, and files. The operating system is the main focus of this layer, as it manages all the other components and provides a communication channel between them. The components that make up the operating system form the set of entities for the model. These entities include programs and threads, files and directories, registries and settings, and other components such as libraries.

Programs and threads being run by the system are easily represented as they have distinct boundaries and are well-understood concepts in literature. In addition, these entities can be directly mapped to representations at the program layer. The program layer shows what each program is doing individually while the host layer ties multiple programs together to represent their combined effect. This type of direct mapping between layers in the hierarchy is useful because it allows the system to correlate information from multiple layers to develop a "big picture" view.

Files, directories, and other components of the file system are also obvious choices to be represented as entities. They provide storage for the computer system and hold results and

information to be used by programs or the end user and can often be the targets of malicious software. Representing each individual file is too complex and storage intensive, so only representing directories or files accessed or modified frequently might be a reasonable solution.

Registry and operating system settings are also important factors to be represented as they often dictate how the operating system performs and functions. Microsoft Windows uses a special directory known as the registry to store settings such as start up programs, hardware profiles, preferences [24]. Making changes to the files or "keys" in the registry can have a major impact on the system, and is therefore a common target for malware. However, many programs or user actions also access and modify the registry. Like files, representing individual keys might be unnecessary and it might be sufficient to group similar keys or entire branches of the registry.

The links between the entities follow the same basic format as the previous layers: read, write, and execute. However, because the operating system uses API and system calls to manage and communicate between components, these calls are used as relationships to provide more meaningful and exact linkages. As described in the program layer, system and API calls provide well-defined functionalities, making them ideal candidates for relationship edges. Using separate links for each call might not be necessary, as similar system calls could be grouped by their general functionality.

Another aspect of the host layer that requires separate discussion is the representation of communication sockets which are used to connect the hosts across a network. Communication sockets are interfaces between programs and the network protocol stack and consist of a local and a remote IP address and port number and a common protocol. Conceptually, they are a mailbox for a program, with the IP address corresponding to the street address, the port number

being the apartment number, and the protocol as the language of the letter. Sockets are managed by the operating system through the use of system and API calls. Sockets are noteworthy because they provide the connection between hosts and could be conceptually drawn at the network layer. However, because the operating system manages them directly through API and system calls, it makes sense to also represent them as entities at the host layer.

The network layer ties together multiple hosts across a network and models their interactions. Individual hosts communicate through connections and packet streams across networks made up of hubs, routers, and other network devices. Defining entities for this layer is more difficult than the previous layers because of the lack of distinct boundaries. Representing individual hosts as separate entities is a logical decision because they are well-defined and can be linked directly to the previous layer. However, representing packet streams and connections is not as simple. Connections between systems could potentially be represented either as distinct entities or as relationships.

Connections are represented as a relationship between hosts with attributes including the port number and protocol. Port numbers and protocols are important because they provide an indication of what type of information the connection is carrying. For example, web requests generally go out to either TCP Port 80 or 443 using HTTP or HTTPS protocols. Requests going to these ports but not using the proper protocols could be an indication of an error or attack.

As discussed previously, sockets are represented at the host layer and are assigned to individual programs. However, the design for the network layer is to abstract away programs that are using the sockets and instead focus on the host-to-host connections. If the system finds suspicious connections at the network layer, it could check the host layer to see what programs

are responsible for the connections. Combining data from both layers could potentially provide data that is greater than the sum of the individual parts.

Table 1 shows the proposed list of entities and relationships which provide the basis for modeling at each layer of the hierarchy [25]. It should be noted that the top three layers have entities that have their own corresponding lower-level model. The network layer is composed of multiple hosts, each of which has an associated set of programs and threads, themselves having their own model at the corresponding layer. This interconnectivity between the layers can be used to either abstract information being passed up, or drill down to determine what combination of low-level actions is creating a specific high-level behavior.

Table 1. List of model entities and their relationships [25]

Layer	Entities	Relationships
Network	Hosts and subnets	Connections (including port and protocol)
Host	Programs/threads, files and directories, registries and settings, libraries, sockets	Read/Write/Execute and API/System Calls
Program	Instruction blocks, data structures, API/System calls, registers	Read/Write
Physical	CPU, memory, I/O devices and storage, etc	Read/Write

3.4.2 Malware Kernel

The model was designed with multiple levels of abstraction to allow the kernel to perform analysis based on how entities fit in the entire world model rather than looking at them in isolation. Having multiple layers allows the system to make decisions at the layer of abstraction most useful for a given situation. It also allows the system to correlate observations across the different levels. Information from the lower level that does not correspond with

observations and predictions made at a higher level could indicate a measurement error or an incorrect prediction. This could be especially useful in the malware domain, as many strains of malware perform low-level actions to hide themselves from higher-level detection processes.

Further, examining information from multiple layers of abstraction provides a more complete view of what is happening in the environment or system. Data that is meaningless at one layer might be extremely useful when combined with information from another layer. For example, a program generating a large number of outgoing connections is not particularly meaningful until it is combined with a large number of similar connections being observed across a network. Combining these two pieces of information could be used as an indication of a worm propagating over the network, a conclusion that could not be drawn by either piece of data by itself.

In addition to the bottom-up benefits described above, using a hierarchical approach also allows a top-down feedback loop to be added to the model. Decisions made at higher levels of abstraction can drive low-level sensors to gain specific information for classification. For example, if a system administrator watching network traffic suspects a worm is propagating through the network, this information could be used to tell host machines to analyze any processes accessing the network for abnormal behavior. This newly acquired data would aid the administrator's decision of whether a worm is truly present. Further, information from the upper levels could be fed down into the system to affect the normal bottom-up processes of forming compound qualia, providing a way for the system to adjust its observations and decisions at individual layers based on higher level context.

Using the previous example, if a worm is suspected at the network layer, host level processes could be adjusted to more likely classify border-line suspicious behavior as malicious.

Specifically, any programs or processes performing unusual network accesses would be examined more closely. Likewise, any processes deemed to be suspicious at the host layer could be examined again at the program layer, adjusting decision boundaries to incorporate the data from the upper layers. While this is a simple example, the ability to feed information up and down the hierarchy to influence and drive both the sensors and decision-making processes offers a potentially powerful engineering advantage over current techniques.

3.4.3 Malware Proof of Concept

A proof of concept implementation demonstrates how a hierarchical model and kernel effectively detects malware. This implementation shows that abstractions of a program and can determine if it is malicious. Using these concepts, an advantageous hierarchical model of qualia is possible in the malware domain. These abstractions can be used for both current techniques and future qualia-based models and kernels.

3.4.3.1 General Design

The basic architecture places sensor input in the first level of the model, which is combined into a second level that represents the concepts and abstractions the kernel will process to form a decision. The proof of concept implementation focuses on the program layer of an individual host. Since the program layer is most often examined by current malware detectors, it will provide a better comparison to other implementations. Additionally, there are many tools that process individual programs and their opcodes, reducing the overhead for developing the implementation. Opcode sequences act as inputs to the first level of the architecture because they are a very low level of abstraction and can be combined to form any number of functions. These functions are the higher-level entities that form the second layer of the model. The kernel will be applied to the second level and used to make decisions on whether the programs are

malicious or benign. If the kernel can properly classify the programs without examing the lower-level opcode sequences, that demonstrates that the higher-level entities capture meaningful information.

Having established an architecture, a methodology is needed to populate the layers of the model from the sensor data. As discussed in Section 2.5, LDA offers a way to find the desired model entities. LDA literature contains algorithms for autonomous parameter estimation and can generate suitable hierarchical models [20]. LDA's clustering mechanism combines common occurrences at the sensor level into new concepts at the second level, which the kernel uses to perform its analysis.

To use LDA for clustering, the words, topics, and documents must be slightly adjusted. LDA in its basic form is a unigram mixture model, meaning that word ordering and sequences are not considered. Word sequences, timing, and other temporal or spatial relationships are important considerations for any qualia-based model, and must be accounted for. For example, changing the ordering of computer instructions changes the output of the program. To account for this, the LDA word vocabulary must accept k-gram sequences of words. At the program layer, k-gram sequences of instructions are the vocabulary for the program documents.

The k-gram sequences correspond to blocks of instructions that perform basic operations such as moving data structures. Latent topics correspond to combinations of these blocks that perform more complex tasks. Combining a basic block that opens and reads a file with a basic block that writes to a new file would provide the complex behavior of copying a file. Extending the LDA method to define chains of entity-relationship-entities as k-grams would allow the grouping of common relationships between entities into topics which become entities at the higher abstraction levels [26]. These entities have their own associated relationships and when

combined with sensor information provide a more complete model. Continuing the example above, the complex behavior of copying a file could be combined with a DVD-burner that indicates the user is backing up their system files to a DVD. This functionality abstracts away implementation details, focusing only on the purpose of the program. For this proof of concept, however, LDA is used to generate mappings from the sensors to a single level of abstracted entities.

Once the LDA process has generated a model for each of the documents consisting of the topics, a decision-making kernel is applied to the results to determine whether the programs are malicious or benign. The authors of the original LDA paper used Support Vector Machines (SVM) as a decision-making kernel; therefore it is also used for this proof of concept consistency [20]. The SVM kernel maps the programs into a topic-space and creates a hyperplane separates the programs into malicious or benign. Even though SVM is not a qualia-based kernel, it still examines the LDA-generated model and make decisions, demonstrating the effectiveness of the model. Figure 3 shows a simple diagram of the proof of concept architecture.

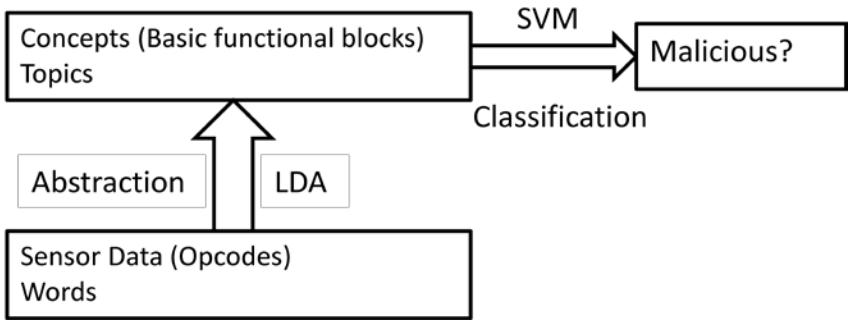

Figure 3. LDA proof of concept diagram

3.4.3.2 Proof of Concept Results

Initial testing and sensitivity analysis are conducted on the proposed proof of concept implementation to determine the best parameters for the LDA algorithm. Sequences of up to ten

opcodes are chosen as words for the LDA model, with a corpus of documents consisting of 300 executables from the System32 folder of the Windows XP operating system and 624 malicious executables from the VX Heavens malware database [27]. Ten opcodes are a reasonable starting size for basic functionality blocks, providing over 150,000 reoccurring opcode sequences as a word dictionary. However, opcode sequences as short as two or three opcodes may be sufficient for a classification methodology [28]. Opcodes sequences are extracted from the programs by using a modification of the Instruction Counter add-on for the IDA Pro disassembly and debugging program [29]. The add-on originally parses the program and counts the number of times each opcode appears, but was modified to count opcode sequences. This research differs from [28] in that it uses not only the individual opcodes but also sequences that occur together in the same program. This provides an additional layer of context not available with individual sequences alone. Any opcode sequences that are not present at least 10 times in the program are discarded to limit topic generation to common opcode sequences. Initial testing shows 10 observations are reasonable, but this number can be adjusted.

The malware samples from the VX Heavens database are widely available and cover 271,092 malware samples [27]. The library was last updated in 2006, but is still be sufficient for a proof of concept test, as the large number of samples provides a representative set of malware at that time. Five types of malicious programs are used in the corpus: backdoors, viruses, worms, Trojans, and constructors. These categories cover the most common types of malware and provide a test set that is consistent with malware observed on the Internet.

Blei's LDA program clustered the sequences into 200 topics representing basic program functions and provided a reduced feature set that can then be used with a number of classification techniques. Sensitivity analysis indicates that 200 topics provides reasonable performance and

leaves enough flexibility for the LDA process to be repeated, thereby creating a topics-of-topics hierarchy for future testing. The corpus of programs was classified into benign and malicious programs using SVM Light package for Support Vector Machines with 10-fold validation. The SVM correctly classified 81.6% of the programs with a 5.6% standard deviation and the results of the individual trials are shown in Table 2 [30]. Precision is the number of true positives divided by the sum of both the true and false positives, and recall is the number of true positives divided by the number of actual positives.

Table 2. Initial Proof of Concept Results

	Overall	Precision	Recall	Correct	Incorrect	Total
Trial 1	89.25	100	66.67	83	10	93
Trial 2	78.49	85.71	40	73	20	93
Trial 3	83.87	100	50	78	15	93
Trial 4	82.8	100	46.67	77	16	93
Trial 5	68.48	100	3.33	63	29	92
Trial 6	82.61	93.75	50	76	16	92
Trial 7	80.43	100	40	74	18	92
Trial 8	79.35	92.31	40	73	19	92
Trial 9	84.78	94.12	55.17	78	14	92
Trial 10	86.02	100	56.67	80	13	92

Further sensitivity testing determined whether changing parameters and using a larger data set improves performance. Table 2 uses opcode k-grams of length ten, a starting LDA alpha value of 0.1, and 200 topics to analyze 924 programs. The size of the k-grams is reduced to five with no degradation of performance, consistent with [28]. The smaller k-grams results in just under 50,000 distinct words for the LDA vocabulary. The number of topics is reduced from 200 to 100 to be consistent with Blei's original work. Finally, the number of malicious programs was increased to 1701 programs that IDA Pro was able to successfully parse, providing a larger data and test set. After these changes, the classification results improved to 90.1% with a standard

deviation of 1.68%, making the change in detection rate statistically significant. The full results of the 10-fold validation are shown in

Table 3.

Table 3. Refined Proof of Concept Results

	Overall	Precision	Recall	Correct	Incorrect	Total
Trial 1	92.5	98.82	98.82	185	15	200
Trial 2	86.5	87.43	98.24	173	27	200
Trial 3	91.5	91.8	98.82	183	17	200
Trial 4	91	90.86	99.41	182	18	200
Trial 5	89.5	89.84	98.82	179	21	200
Trial 6	90.5	89.95	100	181	19	200
Trial 7	89	89.36	98.82	178	22	200
Trial 8	90.5	90.37	99.41	181	19	200
Trial 9	89	89.78	98.24	178	22	200
Trial 10	91.04	90.91	99.42	183	18	201

While these results are not ideal, with approximately 9% of programs being misidentified, they show that LDA can cluster opcodes into a reduced feature-set that allows basic classification in a cyber domain. Replacing the SVM method with a qualia kernel should show improvement in the classification rate as a qualia-kernel would use information from both levels. The proof of concept model is limited to only two levels of entities with limited relationships between them. A more robust model with multiple relationships and levels would offer more opportunities for exploitation by the kernel. Even so, the LDA methodology provides a way to automatically generate the hierarchical and compound entities and relationships needed for a qualia-based model. A model that can be trained with unlabeled data can reduce the need for human analysts in malware detection and allow current defenses to keep pace with the increasing amount of malicious software.

3.5 Summary

This chapter describes the methodology used to develop a qualia-based malware detection system. It outlines the requirements of a qualia system and uses malware detection as a driver problem to design a hierarchical, qualia-based model that can be exploited. A LDA proof of concept demonstrates that concepts can be abstracted from malware to populate a model and successfully determine whether programs are benign or malicious.

IV Agent Architecture Design

4.1 Overview

This chapter describes the process and design decisions used to develop a generalized qualia-based architecture. It outlines the overall goals of the architecture and defines *agents*, *data*, *information*, *knowledge*, and *context* in relation to the architecture. The chapter then discusses the development of the model and kernel for individual agents as well as a system that uses multiple cooperative agents to generate a qualia-based plausible narrative of the world. The chapter closes with an analysis of how the architecture fulfills the requirements outlined in Section 3.2.

4.2 Design Goals

One of the goals of this research is to create a malware-detection architecture that provides an engineering advantage over current methods of identifying malicious software. Current techniques of malware detection are limited because they attempt to classify software based on opcodes, control flow, and other low-level constructs without considering the overall context of the system. Human analysts do not detect and identify malware in this way. Rather, they look at overall behavior and malware interaction with the host system as a whole. If an automated system could replicate this type of analysis, it could better identify novel malware. QUEST's strategy is to create a system capable of holistic analysis through the use of qualia.

A qualia-based architecture is generic and robust enough to be adaptable to a wide variety of application areas with only minimal changes. Humans and other living organisms can perform a wide variety of classification and decision-making tasks using the same physical brain tissue. Therefore, a single, generic architecture that is capable of solving a multitude of current problems should be achievable. This architecture does not have to be an exact replica of what

humans and other creatures use, but must be able to perform the same type of computations in a manner that provides an engineering advantage.

4.3 Defining Stimuli, Data, Information, and Knowledge

A basic qualia architecture to perform classification or decision-making uses qualia to form a representation of the world based on observations and knowledge. These entities reside an environment and are bombarded with various *stimuli*. Examples include wavelengths of light, sound waves, and other physical phenomenon. These stimuli are common to all entities residing in the same environment. Each individual entity has sensors which measure a subset of the stimuli and convert the physical measurements into a format suitable for processing. This new format is the entity's *data*. Unlike the stimuli, data is unique to each entity as it is dependent on that entity's sensors and how the entity uses those sensors.

Data is used to modify an entity's internal representation of the world. Any reduction in uncertainty in the environment is defined as *information*. The internal representation is made up of qualia and the relationships between them. Possible qualia representations are discussed in Chapter V. Like data, information is unique to each individual entity, and is based on an entity's individual data and internal representation.

Whenever an entity converts stimuli to data or data to information, it employs *knowledge*. Knowledge maps function from the stimuli or data input to the corresponding data or information output and can be genetically known or learned/trained from prior experiences. Sensor knowledge (K_s) maps stimuli to data while data knowledge (K_d) maps that data into information. Because knowledge is dependent on the sensors, representation, and prior experiences, it is unique to that individual entity. This means that data, information, and

knowledge are all subjective and two entities presented with the same stimuli will likely create two entirely different internal representations.

4.4 Design

One of the main design goals is a scalable architecture that can be as large or as small as a domain requires. Additionally, a scalable architecture allows the system to grow as it observes new data and patterns. This is especially helpful in problem domains like malware detection where the target changes quickly over time, requiring the system to be able to learn and adapt to keep pace. Scalability can be achieved by creating a network of cooperative generic agents. Each agent shares the same basic structure of inputs, outputs, and internal makeup, but may have different sensors, internal representations, and knowledge. Using a generic structure for all agents allows agents to be added or removed without having to redesign the entire system to accommodate the changes.

A multi-agent design also fits the theorized link-based nature of qualia, whereby individual qualia are combined into larger, more complex compound qualia. For example, consider an architecture in which each agent represents a single concept or quale, with its output representing how strongly that quale is being evoked. Multiple agents can be combined to show the relationships between them or to drive higher-level agent representing a compound quale. Compound qualia agents use regular stimuli and/or the outputs from lower-level agents as additional inputs. This is a form of abstraction; each agent does perform the calculations of the ones below it, but can instead use the preprocessed result. By eliminating the need to repeat the calculations, it reduces the complexity of the higher-level agents.

Further, a hierarchical architecture can be used to influence lower-level agents based on contextual information from higher levels of abstraction. If a higher-level quale is being strongly

evoked from either normal stimuli or even lower-level concepts, it can use that result to affect the agents below it. Consider a person entering an office. They expect to see items such as a desk, a chair, or a computer and automatically look for those items given the context of being in an office. While this type of functionality can lead to issues such as confirmation bias, it also provides a way of influencing individual agents based on the overall state of the system.

4.5 Individual Agents

Having established a basic design of the overall system, the architecture for an individual agent is presented. The design of the agents is based on the description of entities. Entities take stimuli as input which the agent converts into data and then into information. Agents represent the most basic entity which accepts stimuli, converts it into data, and then to information by reducing the uncertainty of its internal representation which consists of a single quale. While the internal representation of a single agent could be as basic as a single value representing the evocation of that quale, the combination of multiple agents provides a more robust representation. This agent is called an atomic agent because it does not use context from higher-level or lower-level agents. It simply relies on stimuli and its internal knowledge to generate its internal information. The agent provides a part or the entirety of its representation to be consumed by other agents or a human operator. The functionality of a single atomic agent is equivalent to the implementation used to represent the knowledge mapping of the input to output and the knowledge mappings are trained accordingly. Herein, multi-layer perceptrons are used, but the architecture is not implementation specific. Figure 4 shows a diagram of the proposed atomic agent architecture.

4.6 Context

QUEST agents provide an engineering advantage over atomic agents by incorporating context from other agents into the decision-making process and tying the outputs of multiple agents together to form more complex agents. *Context* is defined as information transmitted between agents. Adding context requires additional inputs and outputs to individual agents. Once the conduits for context are added, the internal architecture must generate the context to send to other agents and also injest the context the agent receives.

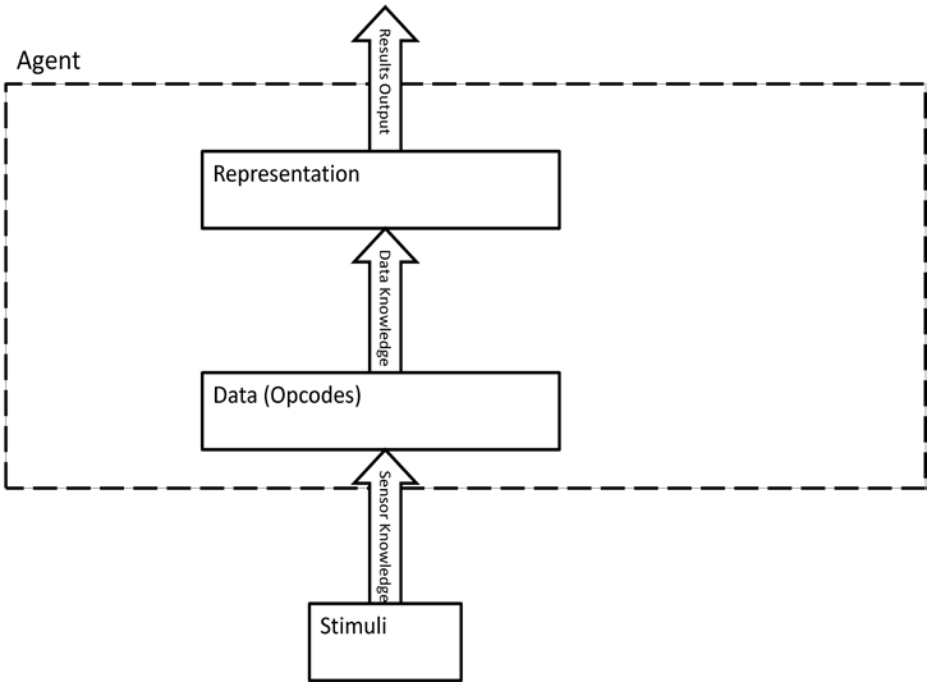

Figure 4. Atomic Agent

Information is unique to the agent that generated it because it is based on that agent's internal representation. Therefore, to transmit information or context, the sending agent (Agent 1) passes portions of its representation to another agent in such a manner that the receiving agent (Agent 2) can interpret and incorporate it into its own internal representation. This requires an

"impedance-matching" between the two agents. Agent 1 must choose what portions of its representation to send, while Agent 2 must use its understanding of the sending agent's internal representation to interpret what it receives. This understanding of other agents is defined as agent knowledge (K_a). It provides a way of mapping the transmitting agent's internal representation in a consumable form, which the receiving agent maps into its own representation based on the origin of the context. Each agent has knowledge of the other agent in the system and uses this knowledge to perform the required impendence matching. Like the other aspects of agents, this knowledge is unique to each agent and can change over time.

To perform this matching, agents only transmit the portion of their representation shared by the receiving agent. For example if two agents both have internal representations of the concept of "chair", then they can share their results of "chairness" as both have a common basis. K_a in this instance refers to how much the receiving agent should trust the results from the sending agent or how accurate that context has been in the past. The receiving agent can adjust its perception to better match the sending agent's. In the chair example, Agent 1 perceives a chair, and the context is passed on to Agent 2. Based on Agent 2's knowledge of Agent 1, Agent 2 can adjust its perceptions to be more likely to locate and perceive the same chair or can compare its own results to the context it is receiving from the Agent 1. This type of shared-representation context allows multiple agents to reach a consensus about the environment and form plausible narratives. It might not be necessary for the receiving agent to be able to independently generate the shared portions of the representation. It simply could rely on the context from other agents to fill that part of its representation. Figure 5 shows the modified agent structure that incorporates context based on shared representations. The "Results Output"

can be reduced to “Context Output”, as both are pieces of the representation being output to other agents, but are annotated separately for clarity.

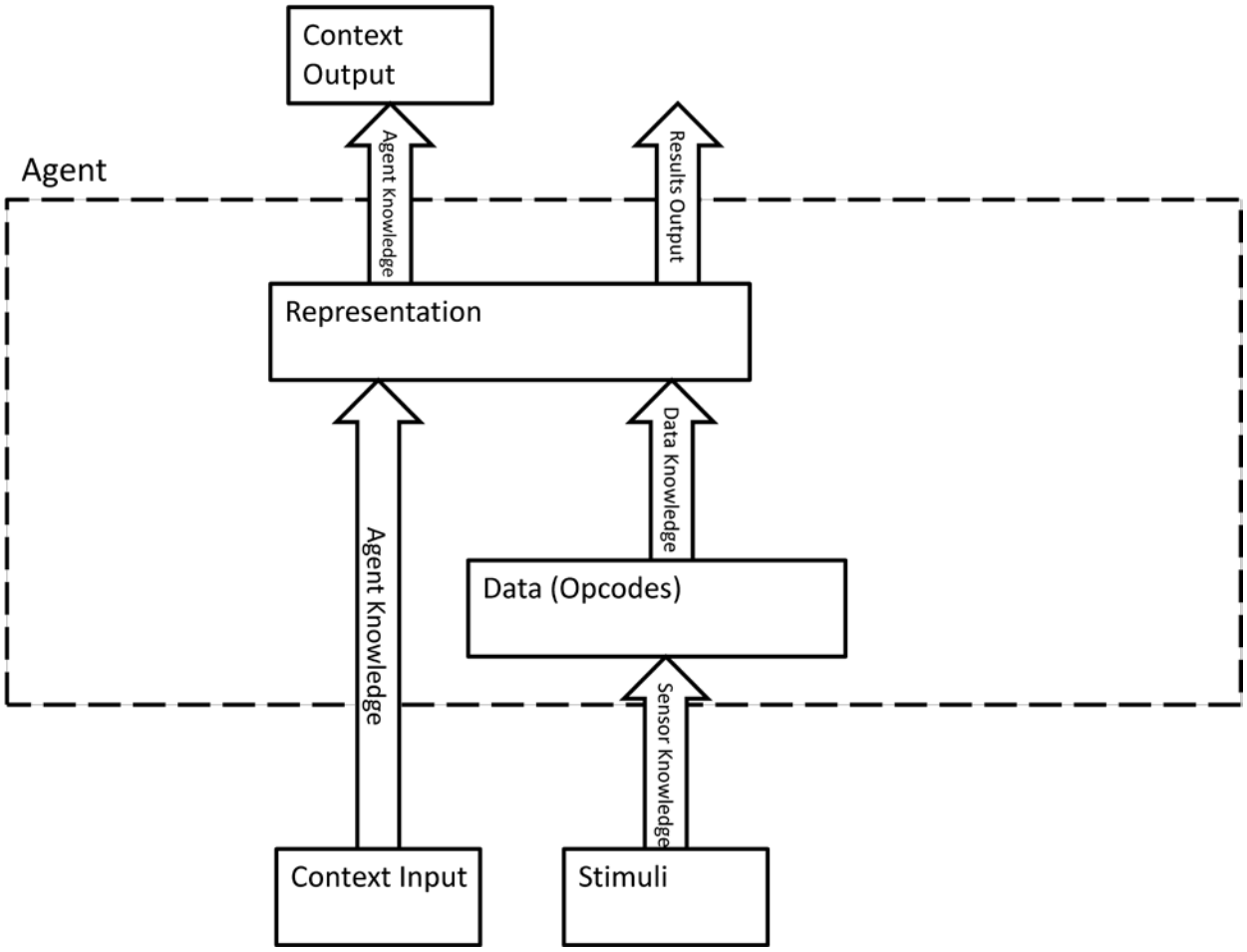

Figure 5. Agent with Shared Representation Context

Another option for context is to take it from other agents as stimuli, allowing the context to affect multiple parts of the representation, not just shared concepts. Continuing with the chair example, if Agent 1 perceives a chair in the environment and then transmits that representation of chair to Agent 2, Agent 2 can use that context to drive its perception of concepts such as “office”, “table”, or “recliner”. This type of stimuli-context allows an agent to impact another agent’s internal representation even if the two do not a share common representation.

Conceptually, stimuli-context is a bottom-up process in which agents at lower-levels of abstraction feed context to agents operating at higher levels of abstraction. An example of this type of structure is shown in Figure 6.

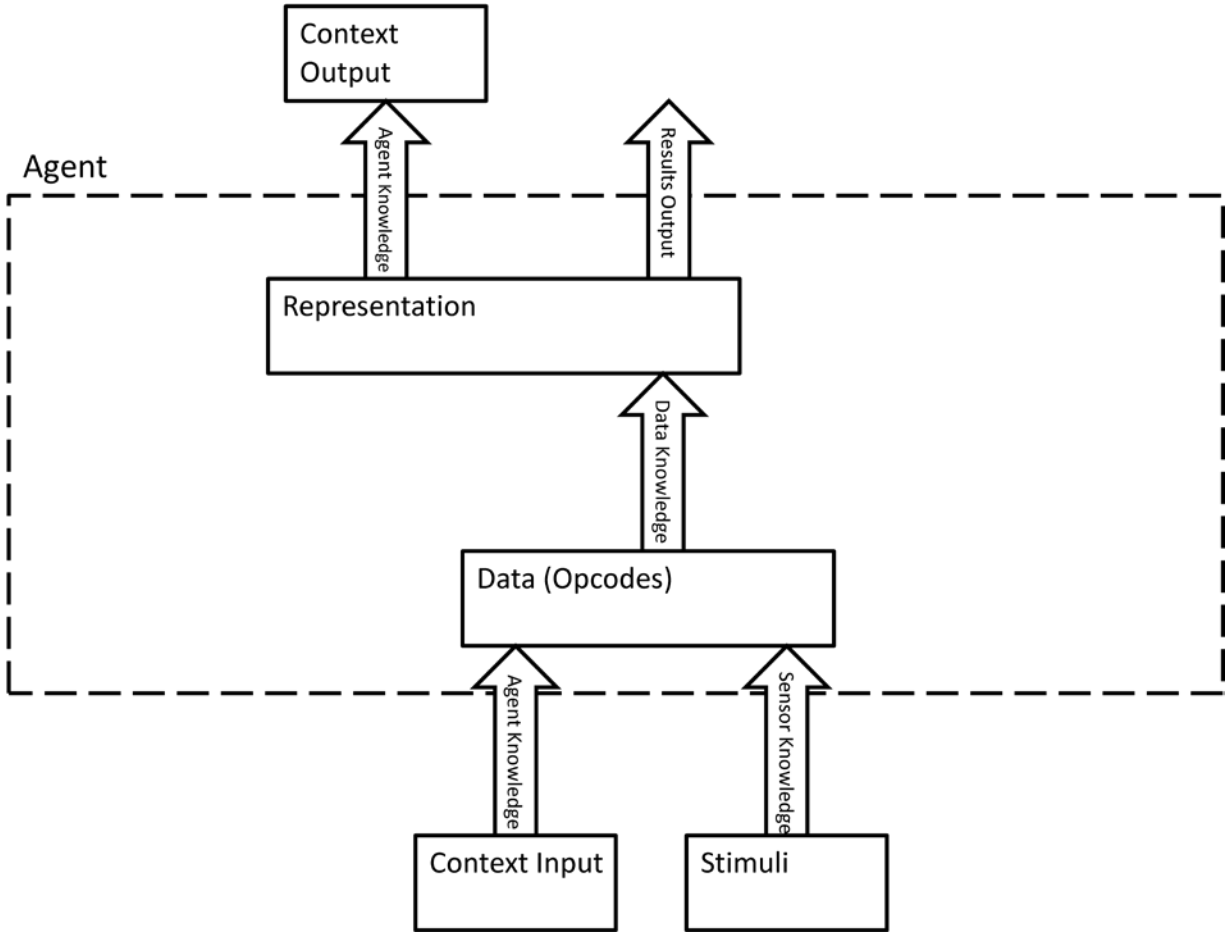

Figure 6. Agent with Stimuli Context

Agents can benefit from both types of context as they perform different functions. Shared-representation context provides a way to check an agent's representation against other agents if the two share common representations, while stimuli-context allows agents to incorporate the results of other agents into account even without a shared representation. An agent with both of types of context can build up compound concepts within its internal representation using stimuli-context and compare it to the results of other agents using the shared-representation context. The resulting architecture is shown in Figure 7. It should be

noted that for simplicity, the shared-representation context and stimuli-context outputs are grouped together into the block labeled “Context Output”.

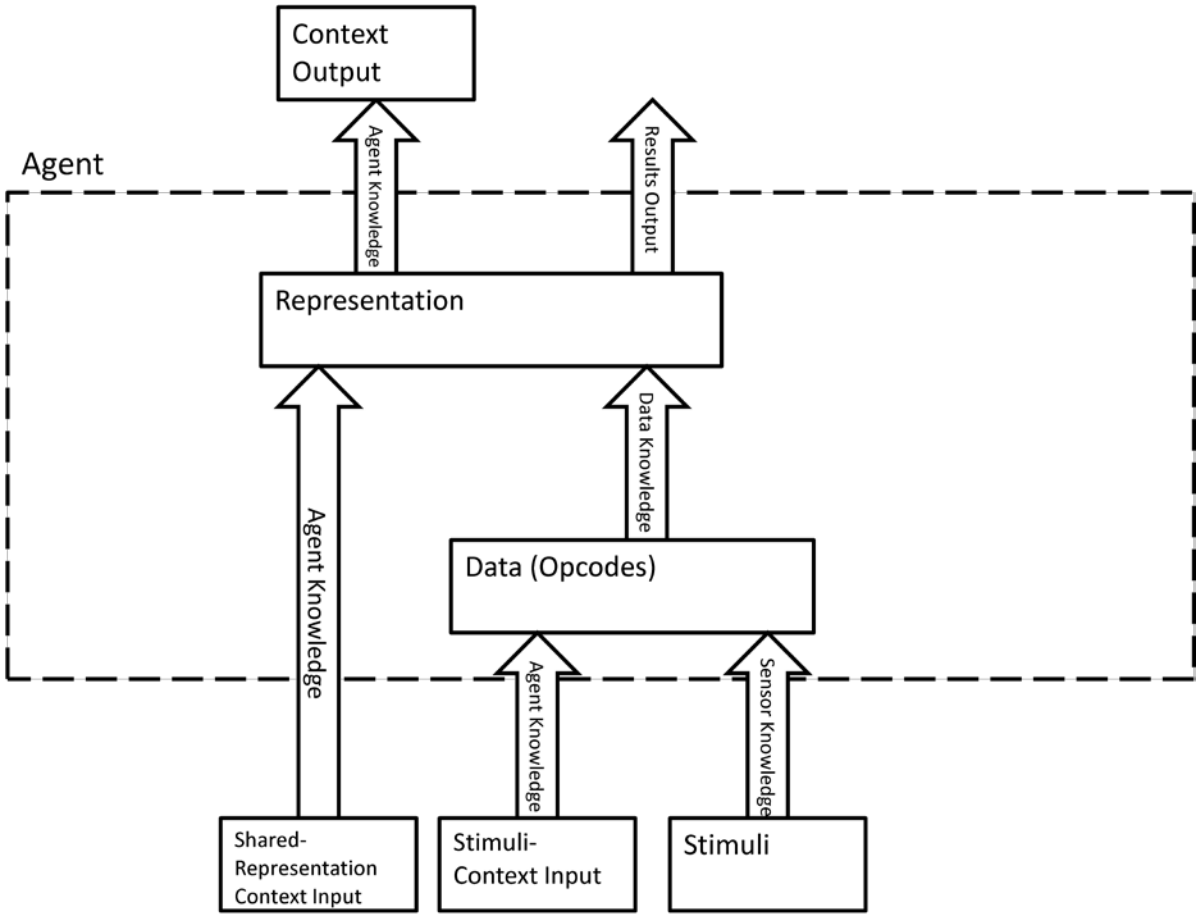

Figure 7. Agent with Stimuli and Shared-representation Context

This also allows agents to demonstrate the QUEST tenets of Libet processes versus Qualia processes. Libettian processes perform instinctive calculations without being brought to the attention or consciousness of the entity [31]. These quick calculations are used to make snap decisions or estimations known as “gists”. For the agent architecture, the bottom-up process and stimuli-context path replicate this functionality. The agent uses stimuli and stimuli-context from lower-level agents to form an initial representation of the environment independent of the results of agents of the same or higher levels of abstraction.

Calculations brought to the awareness of the entity or agent form a plausible narrative based on the interaction of multiple qualia [17]. The subjective nature of qualia means that changes in one quale affect the perceptions of the entity's other qualia. The shared-representation context mimics this process, because it allows agents to affect the internal representations of other agents. To allow each agent to perform its own Libet process but still form a plausible narrative with the other agents, the agent must have two representations: a Libet representation based on the bottom-up process and a Qualia representation that is similar to the Libet representation but is modified based on context from other agents. The different roles of the Libet and Qualia representation are explained in detail in the following discussion of the system architecture. Stimuli-context is referred to as Libet context and shared-representation context is referred to as qualia context. Figure 8 shows the modified architecture with the two separate representations and associated context paths.

4.7 System of Agents

The overall system architecture is a hierarchical system of agents. Because of this hierarchy, the hierarchy levels consist of agents at roughly of the same level of abstraction. Agents can share stimuli-context with any agent at a higher level than them and can provide shared-representation context to any other agent in the system, regardless of level. Agents that provide stimuli-context to a higher level agent are defined as child agents.

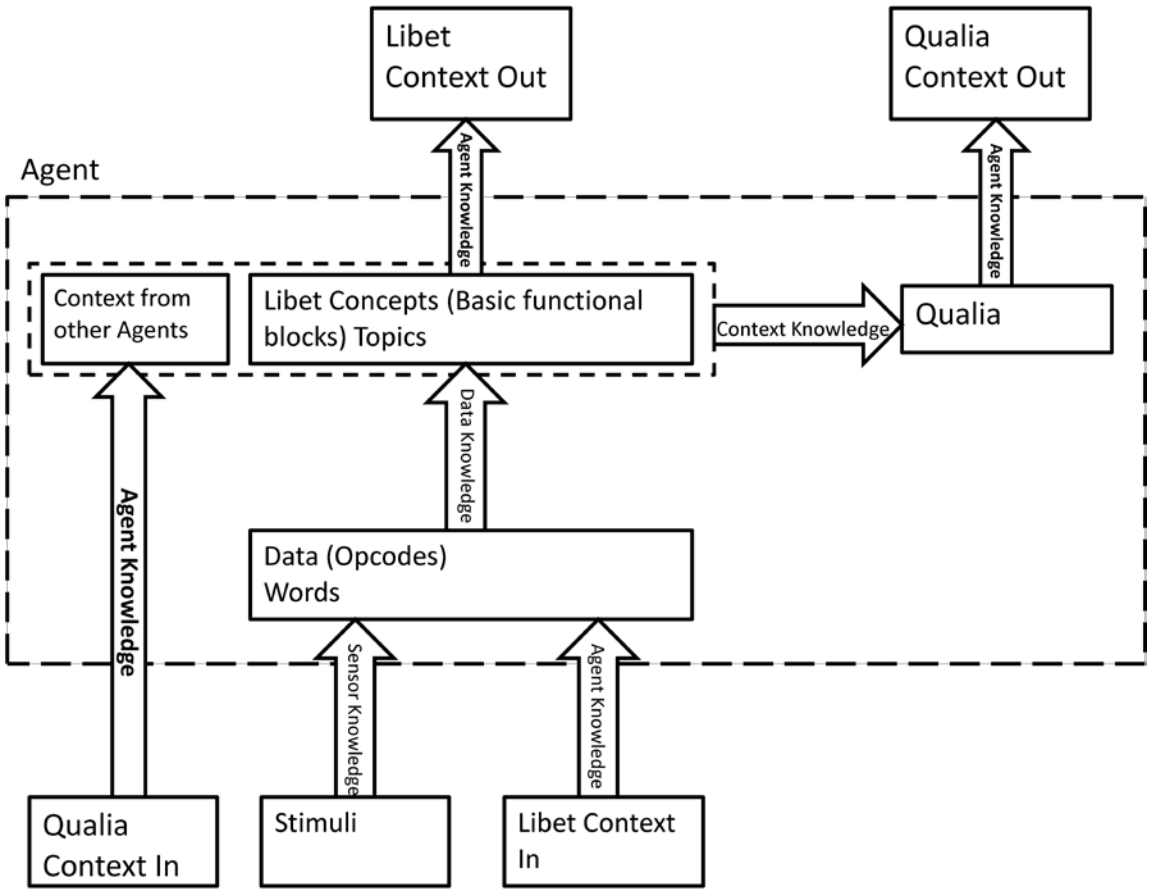

Figure 8. QUEST Agent with Qualia and Libetian Context Paths

Figure 9 shows an example system, demonstrating the context connections and how each agent accepts stimuli from the world. A system of agents can itself be considered a single agent. It takes in a subset of stimuli as data which it uses to modify its representation consisting of the representations of the individual agents. The changes in this representation provide system-level information in the same manner as at the individual agent level.

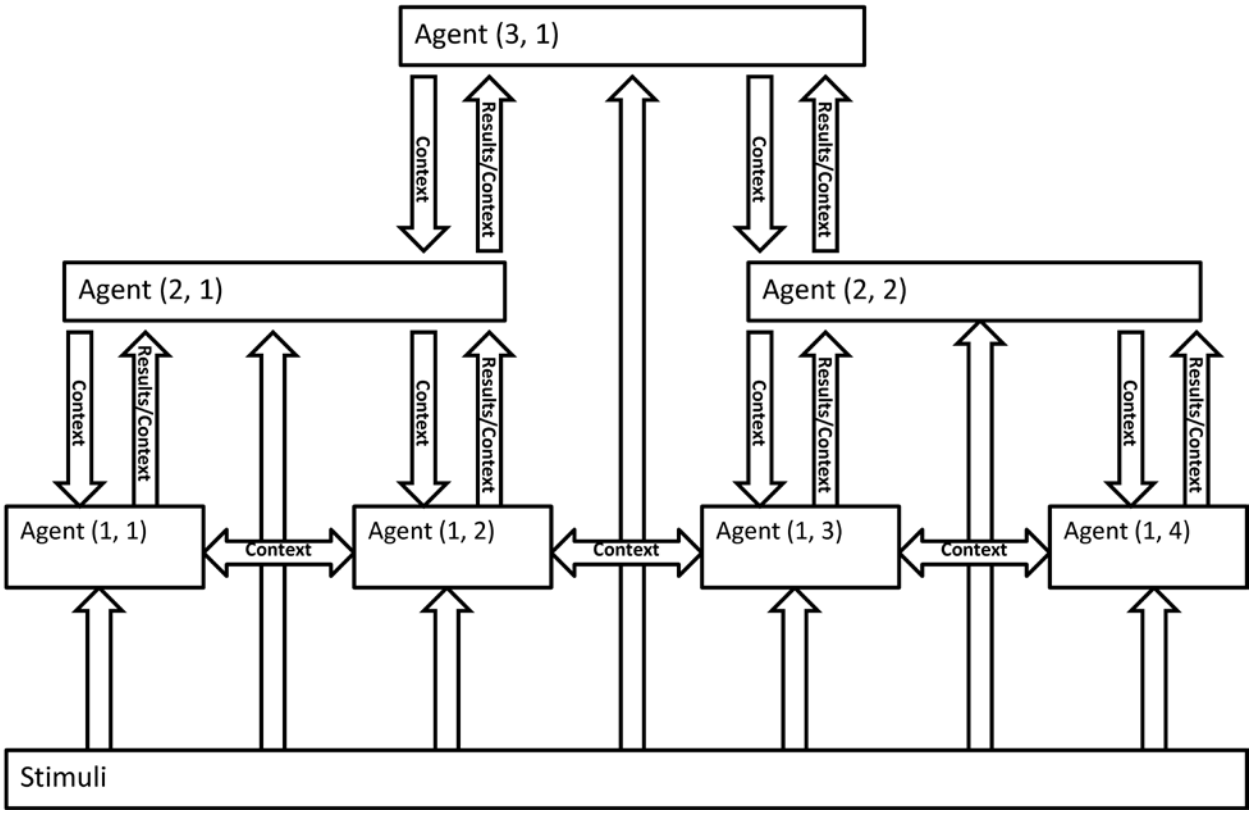

Figure 9. Example of a System of Agents

4.7.1 Libetian Process

The system relies on both bottom-up and top-down processing to make decisions based on the world. The bottom-level agents begin the Libetian process by accepting stimuli as data and processing that into their Libetian representation. The Libetian representation stores the model of the world based on the received stimuli. An example agent, based on a multi-layer perceptron, represents the world with the values of the perceptron nodes, but the representation can be implemented in other ways. The agents export pieces of their Libetian representation as Libetatian context to other agents based on their Ka of the receiving agents. The receiving agents are at a higher-level of abstraction and use both the Libetian context along with their own stimuli to modify their representations. This process ripples up the system, until all the agents have updated their representations based on the world stimuli and the Libetian context from the

other agents. Conceptually, this process mimics a quick estimation or "gisting" of the world with each agent modifying their representations independent of agents at their level of abstraction or higher.

4.7.2 Qualia Process

Once all of the agents have performed their Libetian calculations, the system then generates a plausible narrative where all of the agents interact and affect the perceptions of other agents. At this point, the subjective qualia are generated as the agents' perceptions are no longer affected by their stimuli but are also influenced by the system as a whole. To generate the plausible narrative, each agent modifies its internal representation based on context it receives from other agents. While the Libetian process used stimuli-context, the qualia process uses shared-representation context; agents pass the parts of their representation that correspond to another agent's representation. With this context agents use their context knowledge (K_c) to generate the qualia representation in their internal model. This qualia representation is structurally similar to the Libetian representation but has different values based on the additional qualia context. In effect, the qualia context changes the agent's perception of the world from its sensor-based representation to a representation that is influenced by the context it receives from either the user or other agents.

For example, assume an agent uses a multi-layer perceptron-based representation with five output nodes as its Libetian representation. The shared-representation context consists of other agents' proposed values for the nodes. The agent receiving the context generates its qualia representation by calculating the weighted sum of the context values for each node. The weighted sums are used as the values for the five nodes in the qualia representation.

The agent design allows portions of the qualia representation to be exported to the other agents as context. However, sharing the qualia representations as qualia context limits the system's ability to share context between agents without forming intractable context loops. For example, Agent 1 provides context to Agent 2 which provides context to Agent 3. Agent 3 also provides context to Agent 1. This creates a loop where the context each agent transmits directly influences the context they receive. To prevent these types of loops, Libetian representations are transmitted which remain unchanged during the construction of the plausible narrative. Therefore, they can be shared without the creation of loops. The context each agent receives is independent of the context they transmit, making the generation of the qualia representation tractable.

4.7.3 Narrative Refinement

Once all the agents have generated their qualia representations, the system has generated an initial plausible narrative. Each of the agents has adjusted its perception of the world to be consistent with the results of the other agents. However, the system needs to refine this narrative beyond the initial results. Repeating the Libetian and qualia processes using the new representation is one refinement approach. The system starts at the lowest level of agents who transmit their qualia representation instead of their Libetian representation up as stimuli-context. The stimuli are kept fixed, but the change in the stimuli-context causes the Libetian representation for the receiving agents to change. Beyond the first level of agents, the system has to decide how to handle stimuli-context with the new representations.

The first approach allows the representation changes to ripple up through the system by having each agent transmit the updated Libetian representation as stimuli-context. Only the first level of agents, which do not have a stimuli-context input, uses the qualia representation. They

transmit portions of their qualia representation as stimuli-context to higher-level agents. All of the other agents simply recalculate their Libetian representations based on the changes from that first level of agents and then transmit that new Libetian representation as stimuli-context. The second begins with each agent recalculating its own Libetian representation based on the fixed stimuli and the stimuli context, but rather than sending the changed representation as outgoing stimuli-context, it sends its current qualia representation. The advantage of this approach is that the modifications to the representations remain localized. An agent's qualia representation will only affect the agents that it provides with stimuli-context. In the first method, changes in the lowest level agents propagate all the way through the network, which may or may not be desirable.

Once the changes in the representation have propagated through the Libetian path of the system, the agents can repeat the qualia process by transmitting their newest Libetian representation as shared-representation context. The resulting qualia representations then form the updated plausible narrative. The two processes can be repeated to further refine the plausible narratives. Ideally, the representations would converge to the most plausible and stable narrative. Because the qualia path is independent of the Libetian path, the system is guaranteed to converge in each individual iteration. However, it is possible that the system will not converge to an optimal plausible narrative over the course of successive iterations, as it alternates between different narratives. It is hypothesized that the second method of refinement using the localized update approach is more likely to result in convergence. The issue of convergence is discussed further in Section VI.

4.8 Training Individual Agents

Training individual agents occurs using the training algorithms associated with the knowledge mapping implementations. However, to train each agent, the training set must have labeled data for each part of the representation being trained. For example, an agent with a representation for "chairness" would require training data labeled with a target value of "chairness" corresponding to the input stimuli. Further, the training dataset must not only consist of the stimuli for that agent, but also the stimuli needed by the children of that agent, as they must calculate their Libetian representation to provide stimuli-context to their parent agents. Because this research focuses on multi-layer perceptrons, the agents are trained using back-propagation.

The knowledge mappings associated with the Libetian process are straightforward to train because they rely on a standard mapping from an input to an output. Sensor knowledge, agent knowledge for stimuli-context, and data knowledge can all be trained together. In addition to standard training, the algorithm also adds or removes stimuli from consideration by pruning unnecessary inputs. Training context knowledge is performed separately as it requires shared-representation context from other agents in addition to the Libetian sensor and stimuli-context inputs. Because it relies on both the Libetian representation and the context from other agents, training the context knowledge requires a training set with stimuli for all the agents connected to each agent being trained. It is likely that the training set would require the stimuli for all the agents, depending on the overall connectivity of the system. While the training set requires a large number of stimuli, the number of calculations required to train the context knowledge is limited because the training is only concerned with the context being received and not how that context was calculated by the other agents.

4.9 Agent Connectivity

An important consideration is the connectivity of the individual agents. Figure 9 shows two agents feeding their results upward into higher-level agents, but the possible connections are endless. One method for determining the proper topology is to manually specify them when the system is created. Each agent is responsible for an individual concept and the designer connects agents in a logical fashion based on which concepts feed into each other. Figure 10 shows a simple example of an agent system that examines an office, desks, chairs, and their component parts. These type of expert systems work best for simple, well-understood problems that can be fully defined. However, defining every connection quickly becomes intractable as the problem domain increases in scope and difficultly.

A more scalable and adaptable solution allows agents to dynamically change their connections over time based on how large of an impact the context from the other agent has on the representation of the receiving agent. This requires the system to train with each agent adjusting or pruning its K_a based on an error function of its output given the stimuli and context is receives. Developing a training algorithm is difficult as the flow of context is bidirectional between agents, which leads to looping and possible convergence problems. However, the Libetian context lines are strictly feed-forward and independent of the qualia context paths, making it possible to train the system similar to artificial neural networks. Each component of the agents' internal representation is treated as a node of an artificial neural network and each of the K_a's as a weight. An agent that produces only one concept such as a value of "chairness" would be translated to one node while an agent that outputs multiple pieces of its representation would be represented by a node for each of the outputs. The activation function of each "node" is equal to the activation function of that output, given the agent's current knowledge. Starting at

the lowest-level nodes, the system trains each K_a for the Libet context. In the example shown in Figure 10 the system would start training at the "Desk" and "Chair" agents, training the Libetian context inputs from the "Legs", "Drawers", "Seat", and "Back" agents. Once both the context links for both of these are trained, the system would then move onto training the links to the "Office" agent.

Training the shared-representation context paths is more difficult as shared-representation context goes both directions in the system hierarchy and has a different function than Libetian context in the agents. Libetian context enters the agent similar to stimuli and is mapped to the internal representation based on data knowledge or K_d. Shared-representation context enters the agent and modifies the Libetian-driven internal representation based on context knowledge or K_c. Because the Libetian representation is independent of the qualia, it acts as a bias for each node, with the agent knowledge for the shared-representation context being the trainable weights. The activation function of each node is based on the chosen implementation of the K_c context mapping. As described previously, the creation of the plausible narrative involves transmitting portions of the Libetian representation which is independent of the qualia representation. Therefore, each agent can train its connections in isolation as changes to its qualia representation will not immediately impact the representations of other agents.

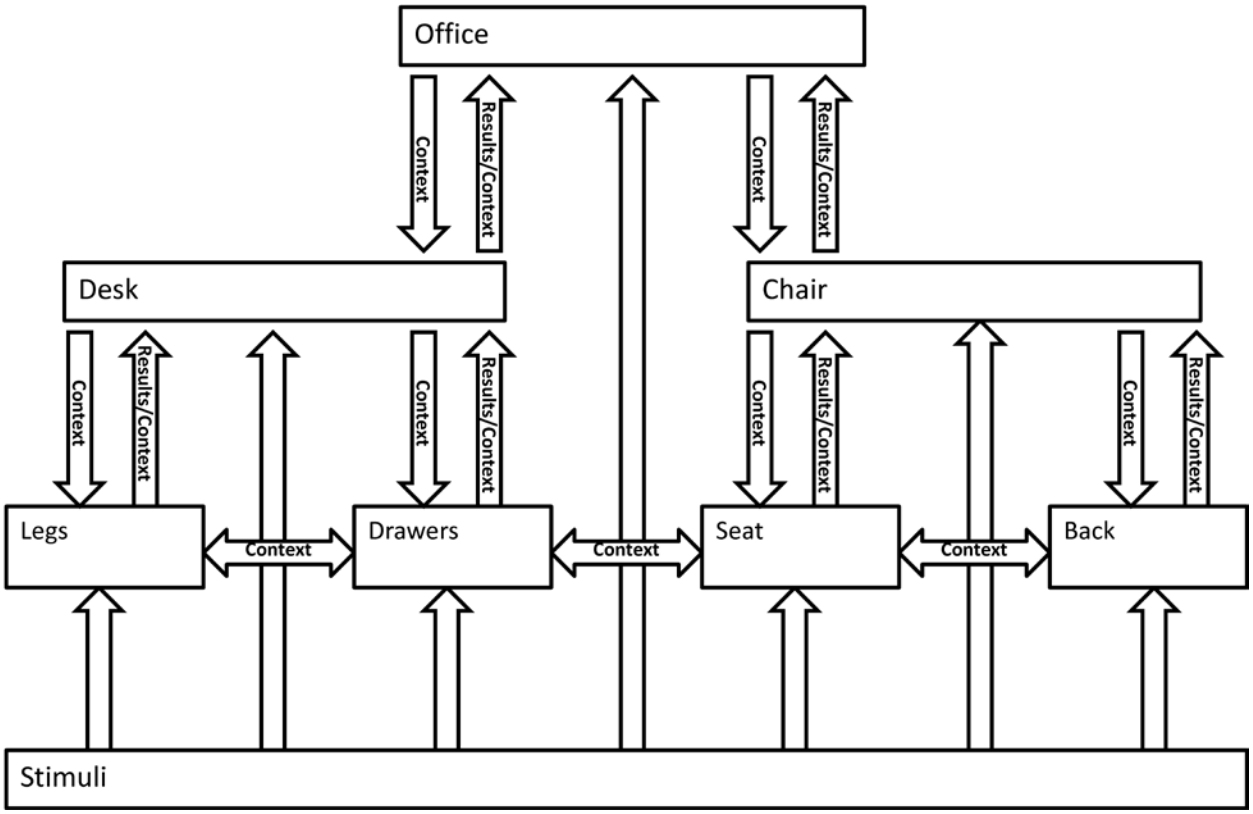

Figure 10. System of Agents for an Office Environment

A final consideration is defining which agents should actually make up the system. Like the connectivity problem, in small systems agents can be manually defined by a domain expert. However, for larger and more complex domains, this is infeasible and an automated solution is needed. In these systems, techniques such as LDA can potentially extract meaningful abstractions and concepts from a level of data, providing the system with candidate agents for the next level of abstraction.

4.10 Results

The architecture described above represents a new approach to tackling pattern recognition problems. Instead of having one main model and kernel for the entire world, the system uses an architecture of distributed agents with each one having its own internal model

and kernel. Each of these agents independently updates the representations of their world model using their personal kernel, sensor input, and stimuli context. These internal model representations and associated kernels can be implemented in any number of ways and may even differ from agent to agent, providing a great deal of flexibility. In addition, the combination of all the agents provides the system with its own unique model that has its own kernel for decision-making. This system-wide kernel controls the flow of context to update individual agents and also allows the system to make decisions based on the representations of all the agents. The decision-making kernel can be implemented with any number of techniques.

4.10.1 Fulfillment of System Requirements

The proposed architecture is compared to the requirements outlined in Section 3.2 to see if it provides potential engineering advantages over current techniques. If the system meets these requirements, it would be able to operate in a manner that will provide more robust decision-making abilities and better overall performance than current methods.

4.10.1.1 Identifies/classifies objects based on relationships and behavior

Each agent is responsible for a quale or entity that the system is concerned with and provides a mapping from input data to a representation of that quale. This determines if a quale for that entity is being evoked in the system, allowing the system to identify those entities in the world. In addition, the system can use multiple agents to identify objects that have several qualia associated with that object. Likewise, an object that evokes multiple qualia could also be classified into several non-mutually exclusive categories. For example, a Honda Civic would evoke the qualia for “car”, “import”, and “compact car”.

4.10.1.2 Uses information from multiple abstraction levels

The hierarchical structure of the architecture allows the exploitation of information from agents at all levels of the system. Each independent agent provides a piece of representation that can be queried or processed by the system-level kernel to make decisions. Further, each agent uses information from not only its own level of abstraction and associated stimuli, but also takes context from agents above and below it in the architecture.

4.10.1.3 Incorporates provided knowledge and learns from experience

The system can have its architecture, agents, and knowledge mappings set manually during initialization. It can also update the agent-level and system-level mappings through training as new stimuli are observed. Each agent can train its individual mapping from stimuli to a qualia representation independently. The system can also modify the connections between agents at the system level, changing how agents affect each other based on how helpful past context was in refining the plausible narrative.

4.10.1.4 Can make decisions based on incomplete data or assumptions

Because the system consists of individual agents, it is possible to provide "assumptions" or targeted stimuli to individual agents to see how those changes affect the rest of the system. Also, if new stimuli are not available for all agents, then the system only updates the agents receiving their stimuli.

4.10.1.5 Predicts future behavior and events based on past behavior and events

This version of the architecture does not model time explicitly, meaning that it has no predictive component. It looks at the immediate snapshot of the world. However, internal representations and kernels for individual agents that do incorporate time and sequences of events can be created. Such a representation and kernel could make predictions and validate

them through training. The Libetian and qualia processes at the system level would function in the same way, simply passing the new representations as context.

4.10.1.6 Weighs multiple predictions against each other

The system can evaluate multiple qualia to determine which ones are being evoked at a given instant of time. Thus, the system can determine which quale is being evoked the most, providing an indication of the most likely quale given the current stimuli and world model.

4.10.1.7 Generates a plausible narrative of what is happening in the world

Because the agents reconcile their individual representations with the other agents in the system, they form a plausible narrative. Their updated representations attempt to fit with the rest of the system's agents, forming a more consistent representation of the world. As agents receive new stimuli, the changes in their representations directly affect how the other agents perceive the world, altering that plausible narrative.

4.10.2 Additional benefits

In addition to fulfilling QUEST tenets, the system architecture also has additional advantages. Because each agent is independent, they can be trained separately, lessening the training overhead of the system. If new training data is received for a single agent, only that agent has to be updated, rather than the entire system. The children of the agent being updated can be trained if desired, but it is not mandatory as in other techniques. Further, an independent agent approach provides a high degree of parallelization in hardware. While the architecture can still be implemented iteratively if needed, a dedicated hardware system could allow it to operate significantly faster. The inherent parallelization can be exploited through the use of parallel and reprogrammable hardware such as Field Programmable Gate Arrays (FPGAs).

In addition parallelizing the calculations, the independent agents design allows other solutions such as signature-based malware detection to be integrated into the solution. While current detection techniques have their weaknesses, they still provide value in detecting malware and should not be thrown away. These solutions can be added to the system as agents and their results used as stimuli and context to other agents. This technique would be especially useful if the stand-alone solutions were modified to accept context thereby affecting their own output and allowing them to fully integrate into the agent system.

A final benefit to this type of architecture is its ability to provide results that a human is more likely to be able to align their own representation with, especially in an expert defined system. By breaking down the plausible narrative into individual agents that represent concepts which are meaningful to a human, it allows a user to examine the evocations of the agents and have a better understanding of "why" the system is producing its final result. Even if the system fails, if it fails in a manner that is understandable by the user which is an advantage over current techniques.

4.11 Summary

This chapter describes a novel, hierarchical architecture of cooperative agents that share context across multiple levels of abstraction. The perceptions of individual agents are modified by the context they receive to form an overall plausible narrative of what they are observing in the world. This allows the system to mimic the subjective nature of qualia, allowing it to operate in domains such as malware detection, where context is extremely important in making correct decisions. The chapter concludes by applying the qualia-based system requirements to the agent architecture and explaining how it meets those requirements and describing other potential benefits the architecture offers over current techniques.

V Pattern Representation and Kernel

5.1 Overview

The agent architecture defined the general structure of the architecture and how data, information, knowledge, and context interact. Since the architecture is generic the internal implementations of the individual agents was defined. This chapter proposes an internal model and kernel that provides a benefit to the system by providing agents the ability to incorporate time and event sequences and to predict future behavior and events. It applies the model to the Blaster worm to demonstrate the use of the model the malware domain.

5.2 Pattern representation and kernel

A graph track the relationships between entities and a kernel that analyzes patterns to predict future behavior based on past events. The current design is at a very high-level of abstraction, but can be expanded and refined through future research.

5.2.1 Proposed Model

The proposed model is includes entities and the relationships between them [17, 32]. Due to the importance of context and relationships between entities, graph-theory is used to model and interpret them. Entities are nodes while the relationships are directional links or edges between nodes. Graph theory provides standard terminology and algorithms which can be leveraged for this application. Further, there are a large number of tools for working with graphs, which will prove useful.

Nodes represent individual instantiations of entities observed in the environment and are labeled with unique identifiers such as “Bob”, “Lamp 1”, or “Attack 2”. Each node has certain attributes associated with it such as height, length, and weight. These attributes could be used

with standard pattern recognition techniques to provide additional input, or to determine of relationships between objects such as “smaller than”.

Conceptually, each node has a “color” or category/class that corresponds to a basic category or concept such as “person”, “small, movable object”, or “network event”. An entity can belong to a large number of categories or subcategories, meaning each node could have multiple “colors”. Modeling the categories or concepts that an entity is in is important as objects in different classes often behave differently or have different relationships with other entities. These differences can predict future behavior based on an entity’s class and relationships, or be used to classify unknown entities based on their behavior and relationships. The implementation of this coloring is later in the chapter.

Edges represent relationships between the objects such as “on top of”, “inside”, “moves”, “smaller than”, or “before”. These directed edges represent how entities are tied together in the environment spatially, temporally, and functionally. Simple relationships such as “Object 1 is on-top-of Object 2” can be represented with a one-to-one edge. However, more complex relationships such as “Puppet 1 hides Object1 from Puppet 2 inside Object 2” require hyperedges capable of connecting multiple vertices. Each edge has a label specifying the relationship it represents, and can be weighted indicating the strength of the relationship or other information.

Representing the world as sets of nodes and edges allows the association of objects that might not be directly connected. For example, suppose Mr. Smith “lives in” House A, and House A “is located on” Academy Boulevard. While there is no direct connection between Mr. Smith and Academy, the system could follow the links to determine that he lives on Academy. Combining relationships in this manner provides the basis for generating higher level concepts and more complicated relationships.

A graph of the system provides a snapshot of its perception of the world at an instant of time. As time passes, links are added, removed, or modified based on changes in the world. The changes in these links form the patterns described in the previous section. The system stores these snapshots for playback or to revisit past decisions. Care must be taken in setting the size of the time window between snapshots, as too large of a window will miss important changes, while too small of one could lead to inadvertently splitting conceptually simultaneous occurrences into different windows.

Edges form a vital component of the model, as indicateore how an object relates to or interacts with the world. These relationships are one of the major factors in evoking specific qualia. Therefore, categories or node colors are implemented using these edges. A category of objects is the set of allowable edges to other vertices. For example, containers such as boxes or cups could be defined as any object that can have the relationship “is in” directed to itself by another object. If the system sees an object with that relationship, it would recognize it as a container. Further, the system could incorporate the class or attributes of the node at the other end of the edge.

For example, cars are found “on” roads or “in” garages. This would differentiate a car from dishes that are “on” tables or “in” cupboards. Additionally, the system could use anti-edges to represent relationships that can never exist for that class. The system would keep a history of observed edge types for each node to determine its class even if all the relationships are not currently present. Keeping track of observed edges also allows objects to be members of multiple classes, which makes intuitive sense.

5.2.2 Proposed Kernel Design

Having outlined a model for representing the environment, a kernel is needed to process the model, make predictions and classifications, and learn from its observations. The model provides the storage for the system, but the kernel is what provides the system with the ability to "reason".

A form of reasoning is achieved using of "patterns" which store changes in nodes and links. These patterns are either learned or provided a priori. The simplest patterns consist of before and after states, showing the state of a set of nodes and edges at a specified time along with the new state after a set transition time. Conceptually, this type of pattern states "if in State 1, expect State 2 next." Such a pattern is similar to a simple "if-then" statement, but it provides a basic foundation for more advanced and robust patterns. Consider, for example, dropping an object. The initial state is the object being released from the hand and the final state is it coming to rest on the ground.

The kernel has three major components—a set of patterns, a reasoning component for making decisions and predictions, and a learning component that adjusts or creates additional patterns as needed. The pattern set contains both patterns provided as knowledge and patterns learned through observation. These patterns have a before (prerequisite) state and an after (result) state. The initial state is a set of nodes and associated edges that are required to make the prediction and the final state is the expected edges after the transition. Patterns can be multilayered, examining vertices and edges across various conceptual layers, or focus on an individual layer. Patterns can also use both node and edge coloring to account for behavior for specific object classes.

The system's reasoning kernel uses the pattern set to predict behavior or identify and classify objects. In its most basic form, the reasoning kernel takes the prerequisite graphs of a pattern and compares them to subgraphs in each model snapshot. If a match is found, the system predicts the result state will follow in the next snapshot. This approach however, known as the subgraph isomorphism problem, is NP-complete [33]. However, this can be mitigated somewhat by using heuristic search functions. For example, choosing a vertex from the pattern that has a rare color, has the largest number of edges, or some other "rare" attribute and then parsing the world model to find matching nodes to use as a starting point for the search. This issue warrants careful consideration.

The system must make reasonable predictions using both generalized and specific patterns. If the system observes a world state that matches one of the generalized patterns but not any of the more specific patterns, then prediction is simple. However, the prediction becomes more complicated when a set of nodes and edges match multiple patterns. The system must determine which pattern to apply or if both should even be used. For example, if Mr. Smith comes home cold and hungry, does the system predict he is going to eat, that he is going to sit by the fire, or that he does both? One approach would allow only one pattern to predict the addition or removal of a specific edge at a time. If two patterns attempt to modify the same edge, then they must compete to determine which one is used. Perhaps by choosing the one with the highest prediction accuracy or the one that matches the current world model the best.

Patterns provide the system a way to "reason" based on incomplete data or a simplistic version of a "hunch". If the matching requirements are relaxed where certain edges or node colors are not always required, the system can make alternative predictions it might not make otherwise. These potentially have lower associated prediction accuracy, but allow the system to

function even with sensor errors. This type of technique is also important in reconciling failed predictions as the system examines the “what ifs”.

Thus far, patterns have focused solely on changes in edges between nodes. However, for a layered approach, the system must also consider the combination of basic patterns into more complex, higher-level patterns. For example, suppose a layer has patterns for placing an object into a container and distracting a person. At the next higher level, these patterns are represented as (hyper)edges between nodes. These edges, then, are used in the same manner as any other edge and form a new pattern known as “hides”. This a powerful tool since it provides a method for abstraction. The “hides” pattern does not need to know what caused the patterns for “puts in” or “distracts”. It just needs to know that they were evoked. This removes the lower-level details and focuses only on the general concept.

Two final aspects to consider are time and space. Important changes are not required to occur sequentially and form simple before-and-after patterns. Neither will patters always involve a closely located set of nodes. Related events might be separated by large amounts of time or space or even both. For example, a drought in the American Midwest could drive up the price of rice overseas months later. This remote relationship can be modeled by connecting correlated patterns with relationships such as “happens before”, “happens simultaneously”, or “happens near”. Notionally, each of the individual patterns are represented by a node and connected by a relational edge. In essence, it becomes a pattern of patterns. Relationships could span multiple layers, tying together observations at various levels. These nodes/edges are not in the world model, but stored with the pattern set and used only by the reasoning kernel.

The final component of the kernel is the learning module. This module reconciles bad predictions and adapts based on new observations. Ideally, the system operates in a changing

environment with little supervision from a human operator; modifying its patterns or creating new ones as needed.

The system has three basic failure modes—predicting a transition that does not happen, failing to predict a transition that does occur, or predicting a transition that occurs but with incorrect links. These failures can manifest themselves through the absence of expected (missing) links or the addition of unexpected (extra) edges. When a failure occurs, the first step is to determine the failure mode.

If the system made a false prediction, it ignores that prediction temporarily and creates a "difference graph". The difference graph has the extra and missing edges between the prior state and new observed state, along with the nodes they connect. The two types of edges have different notations to tell them apart. The difference graph is a basic representation of what relationships changed. Each pattern has an associated difference graph to indicate what relationships changed that transition.

After calculating the difference graph, the system determines whether a sensor was faulty or an invalid assumption was made by comparing the difference graph to the graphs of each of the patterns. If the difference graph matches a pattern's graph within a certain threshold, then the system could hypothesize that it simply missed something and continue on. It could also determine a pattern was too specific and generalize it by attaching probabilities to certain links or node colors in the prerequisite graph and using the presence of those links and colors in subsequent predictions. Eventually, extraneous edges are removed as their percentage drops to zero. Alternatively, a new pattern that does not have the missing edges could be created. This pattern competes with the more specific pattern in future situations. Since multiple patterns

could match within the threshold, the system uses the closest match rather than all the potential patterns.

If the system does not find any suitable matches, it creates a new pattern. The difference graph provides a starting point as it indicates which nodes had their links change. Creating the pattern simply takes the before and after state of these nodes and their edges. This new pattern is trained over time with new observations.

If the system predicts a transition which did not occur, the pattern is adjusted or the prediction accuracy associated with the pattern is reduced. Reducing the prediction accuracy phases out patterns that do not work as intended or are no useful because of changes in the environment. Adjusting the pattern is more complicated, however, as there is no reliable way to tell what links or node colors are required to make the pattern more specific from a failed pattern. The system could keep track each time the pattern was evoked and compare successful and non-successful predictions, using those differences to adjust the pattern.

The most difficult situation is when the system predicts a transition and was partially correct. Does the system need a more specific pattern to predict this particular instance? Did another, separate occurrence affect the links? Did another transition occur after the one predicted? By default, the system assumes that a more specific pattern is needed and creates it in the same way described above. This pattern is adjusted as more information is collected and eliminated as needed.

In any of these cases, the system can apply the newly adjusted models to past observations and to improve the system accuracy. If the accuracy does improve, it is likely the system has improved its pattern set and can keep the changes. If the accuracy does not improve, or degrades, the system needs to revoke the changes and look for another option.

One of the strengths of this system is it can indicate which patterns are used in the decision-making process—essentially the reason for the prediction. Ideally, these patterns map to relatively understandable concepts, providing more meaningful feedback than "the codebook vector fell outside the decision boundary." Thus, patterns can be manually examined for flaws and adjusted, providing complementary assistance to the system's learning ability.

5.2.3 Prediction Accuracy

To provide additional functionality to the system, information can be added to the patterns. For example, the prediction accuracy of the pattern indicates how often the predicted state follows the observed "before" state. A pattern that predicts the impact point of a free-falling object would likely have high prediction accuracy. Predicting the direction a car is going to turn at a four-way intersection would probably have a lower accuracy. Associating prediction accuracy with each pattern assigns a level of certainty to predictions and provides a metric to weigh competing predictions.

Assume that cars coming to an intersection have a 50% chance of continuing straight and 25% chances to turn right or left. This situation is represented by three patterns, each with the same initial state-a car coming to an intersection. Each would have a different final state-the car going straight, turning left, or turning right, along with the associated prediction accuracy. By adding prediction accuracy to the end states, the system has a metric to improve predictions.

5.2.4 Generalization and Specification

Prediction accuracy of patterns can be used to implement specification and generalization in the system. Generalization uses a number of distinct observations to create a pattern to apply to a variety of situations, even if the measurements do not match any of the original observations. A general pattern finds commonalities between distinct, but similar patterns. Specification is the

converse of generalization; it adds extra information to a general pattern to provide a better prediction. It isolates differences between patterns and uses those distinctions to make different predictions. Generalization predicts behavior in a larger number of circumstances, while specification improves predictions by incorporating additional information.

Continuing with the car example, suppose that cars at the Woodman and Academy intersection turn right with 50% frequency and go straight 25% of the time. The previous pattern was generalized for all intersections to allow the system to predict which direction a car would turn even if the specific car or intersection has not been observed before. However, because cars at the Woodman and Academy intersection behave differently than at intersections in general, the system will not accurately predict their behavior. However, specification can create a pattern to apply to this particular intersection, and accuracy would increase. Similar patterns could be used if a specific car was more inclined to turn left at any intersection it approached or if cars approaching intersections from the north went straight more often.

5.2.5 Multilayered Patterns and Context

Patterns to this point use probabilistic means to make their predictions. Such an approach, while valuable, it is not an intelligent system. It can calculate the probability of cars turning certain directions, but it cannot say why they have that probability. For example, assume that Mr. Smith lives on Academy, just north of the intersection with Woodman. A person observing Mr. Smith approaching the intersection from the south, east, or west will predict that he is likely to turn north because he is probably going home. A person can make this guess without any prior observations of Mr. Smith's driving behavior, because he or she can use intuition to guess at Mr. Smith’s higher-level goal. A probabilistic system would have to see Mr. Smith at that intersection several times before developing a probabilistic pattern for his behavior

because it lacks the ability to model higher level concepts. If the system can be given the knowledge about what "going home" means, then it can use this higher level concept to understand which directions cars will turn.

Representing these higher level concepts as patterns is not trivial as the pattern must capture the desired meaning but be general enough so it can be applied to multiple instances. Simply adding the information that Mr. Smith's house is located near the Academy/Woodman exit to the existing pattern gives additional information to predict his behavior at that single intersection. However, a pattern that follows his car's movement and observes he is getting closer to his home could infer Mr. Smith's ultimate destination is his house. This type of pattern could be used to predict which direction Mr. Smith's car would turn at any intersection if the system infers he is headed home. Further, this pattern could be expanded to work with any destination that Mr. Smith frequents.

The "going home" pattern illustrates several important concepts to develop for the system. The first is the need for multiple layers of observations and patterns for robust prediction. For this example, elements such as the position, speed, and direction of the car make up a basic layer of observations and patterns. The next layer is information and patterns on roads, driving regulations, traffic, and behavior at intersections. Another layer contains patterns and relationships related to locations and businesses Mr. Smith frequents or even information such as the human tendency to spend each night at the same location. This type of hierarchy is valuable because the system can combine information from different layers to predict behavior. Knowing that the position of Mr. Smith's car (information from the lowest level) is approaching his house (information from the highest level) allows the system to predict which way he will turn at intersections (middle layer).

Ideally the layers would be tied very strongly together with patterns and observations at lower levels being combined to form and evoke patterns at higher levels. The combination of patterns forms an abstraction by tying combinations of patterns into larger patterns that are more useful or have more meaning. For example, "Mr. Smith is driving home" conveys more information than a sequence of turns or position changes. By creating more abstract patterns, the system can make decisions or convey information without having to focus purely on physical measurements.

Issues in the "driving home" example are temporal considerations and extended pattern chains. Determining a car is moving closer to home is difficult to determine from a single turn or change in the car's position. However, if the system tracks several turns in sequence, it can better determine the car's eventual direction. Keeping a history of past occurrences and observed patterns can enhance prediction. Extending the "length" of patterns from "before and after" to "first, second, third" provides part of this functionality, but not all events will happen in a sequential manner. Pertinent events and patterns might occur in various orders, simultaneously, or be separated by large periods of time or space. The system must be able to recognize and use these events to form better predictions.

5.2.6 Object Classification

Much of the previous discussion has focused on making predictions of future behavior. However, another goal of the system is to use behavior to recognize, identify, and classify objects and entities. The system could use patterns to keep track of acceptable and normal behavior for classes of objects along with common relationships between the objects could be tracked. For example, if the system observed a previously unseen animal, it could rely on physical descriptions such as size, shape, and color for identification, as many current pattern

recognition techniques do. However, the system could also use behavior and relationships such as chasing thrown balls or wearing a leash to determine that the animal is probably a dog. Going back to the car example, the system could determine whether Mr. or Mrs. Smith is driving the car based on driving behaviors or destinations. What makes this approach interesting is the patterns and relationships associated with objects and classes can be thought of as qualia-like. Part of "dogness" would be patterns and relationships such as chasing balls, being associated with leashes and drinking bowls, and wagging tails.

5.2.7 Reconciling Incorrect Predictions

A final set of considerations is the system's ability to handle incorrect predictions and adjust itself to perform better in the future. One of the major problems with standard pattern recognition techniques is when they misidentify objects, they are unable to determine what caused the misidentification. If the system above examines what patterns it used to predict behavior or classify an object, it could modify the associated patterns or provide rationale for its decision. Suppose the system has decided Mr. Smith is driving south towards his home, but his unexpectedly turns east at an intersection.

At this point the system would determine what caused its turn prediction to fail. It could report it made its decision based on Mr. Smith's previous history at this intersection and its assumption that he was driving home. The system could then examine whether its patterns or assumptions were flawed, additional factors should be considered, or even whether it had received faulty data from a sensor. The system could postulate, "Maybe the assumption that Mr. Smith was going home is false, but he was going to a grocery store in that direction instead." The system could generate several of these plausible narratives and find the one that best fits by previous observations. The system could retrace Mr. Smith's drive from work with the

assumption that he was going to the store instead of home and see if his choice of turns better fit that assumption. The system might also look at information across layers and see if factors not considered might have an impact on the behavior. It might determine that the southbound lane was blocked by construction, and infer that was the reason Mr. Smith turned east. The system would then still use the "going home" pattern as Mr. Smith attempts to drive around the construction zone. If any more predictions failed, then the system could once again go back in an attempt to reconcile the differences with different narratives or assumptions.

5.3 Malware Example

To illustrate how the model works for malware detection, it is applied to the Blaster Worm released on August 11, 2003. The Blaster Worm was a fast-spreading worm that affected systems running the Microsoft Windows operating system. The worm self-propagated to over 100,000 vulnerable hosts via a buffer overflow of the DCOM RPC service [34]. Infected hosts attempted to conduct a distributed denial of service attack against windowsupdate.com by flooding the site with SYN packets. The worm also generated large amounts of network traffic and caused many hosts to repeatedly restart. Because of Blaster's impact, it spawned multiple variants and copy-cats and is well-researched and referenced in literature. Further, it did not use polymorphism, stealth, or other protection mechanisms that make analysis difficult or complex. Therefore, it is a reasonable choice to model as a test of the proposed system [34].

The description below includes all the layers of the model, but focuses on the host layer, as it has the most activity. In this example, three computers are connected in a network: Attacker, Target, and Future Target. Attacker has been infected with the Blaster worm and compromises Target which then attacks and compromises Future Target. Figure 11 shows the relationship between the three hosts. This example focuses on Target and the models only that

system. However, the model would be the same for the other systems, as the worm does not change its propagation method as it spreads. The relationships between the entities are labeled with simple meanings easily determined from the system calls.

Figure 11. Example Setup

At t=0, Target is running the RPC service which listens on TCP port 135 as shown in Figure 12. The RPC socket receives data which contains the buffer overflow sent from Attacker. While this is occurring at the host layer, the network layer observes the connection between the hosts and the program layer follows the operation of the DCOM RPC service thread.

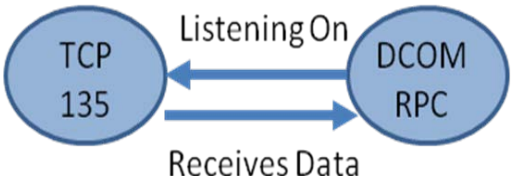

Figure 12. RPC Service Receiving Buffer Overflow

Once the vulnerable DCOM RPC service is compromised by the buffer overflow, it spawns a listening shell port on TCP port 4444 as shown in Figure 13. This port is monitored for incoming commands from Attacker, which it receives in Figure 14.

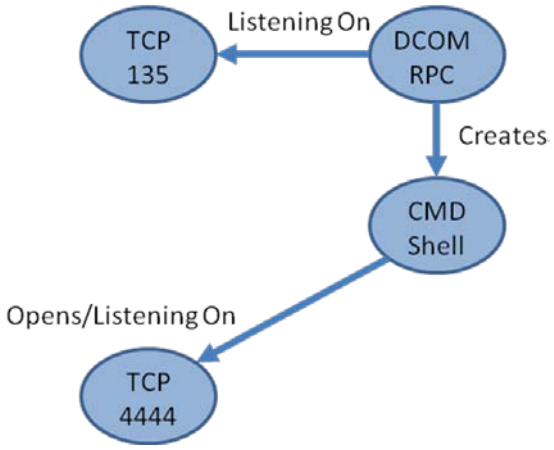

Figure 13. RPC Service Creating CMD Shell Listener

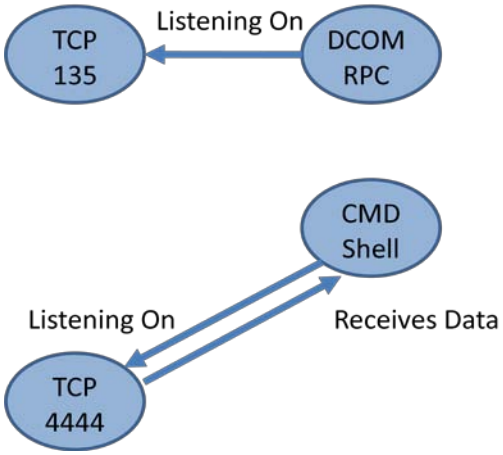

Figure 14. Listener Receive Commands from Attacker

Once the command shell receives data from its socket, it opens a TFTP connection back to Attacker and downloads the actual worm, MSBlaster.exe. This process is shown in Figure 15. Once the data is transferred, the command shell saves the downloaded file and creates a registry key to run the worm whenever the system restarts as shown in Figure 16.

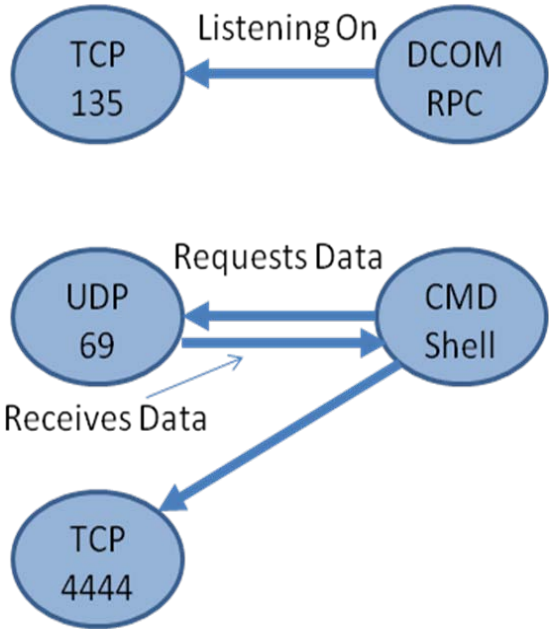

Figure 15. Listener downloading with TFTP

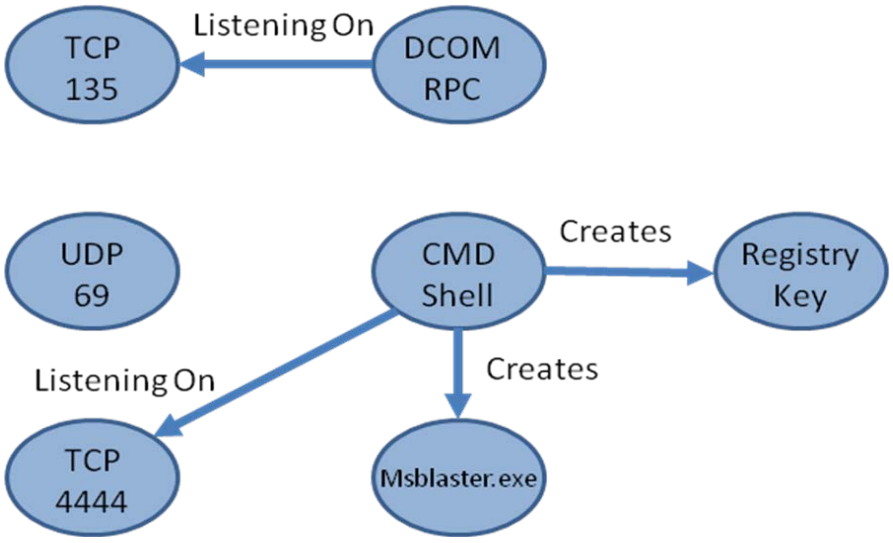

Figure 16. Listener writing MSBlaster.exe and Registry Key

Once the worm has been successfully downloaded and is executed, it begins the distributed denial of service attack to port 80 and attempts to infect other machines by sending malicious connections to random hosts with an open port 135. This is shown in Figure 17. This

full sequence of events could form a pattern for worm-like behavior. However, these events alone are not enough to make a definitive classification as other benign programs have similar behaviors. This ambiguity could be mitigated by adding information from the other layers. The program layer could detect anomalies in the behavior in the RPC service thread or detect suspicious sequences of instructions in MSBlaster.exe, which cast suspicion on the system's behavior. Likewise, observing a marked increase in network traffic on ports 135, 69, and 4444 across the network further reinforces the hypothesis an attack is occurring. Correlating data from the different layers is vital as benign programs could exhibit any of the behaviors, but would be unlikely to produce all of the behaviors at once. It is only by combining the full spectrum of effects and results that an idea of "Blaster-ness" or "wormness" is developed.

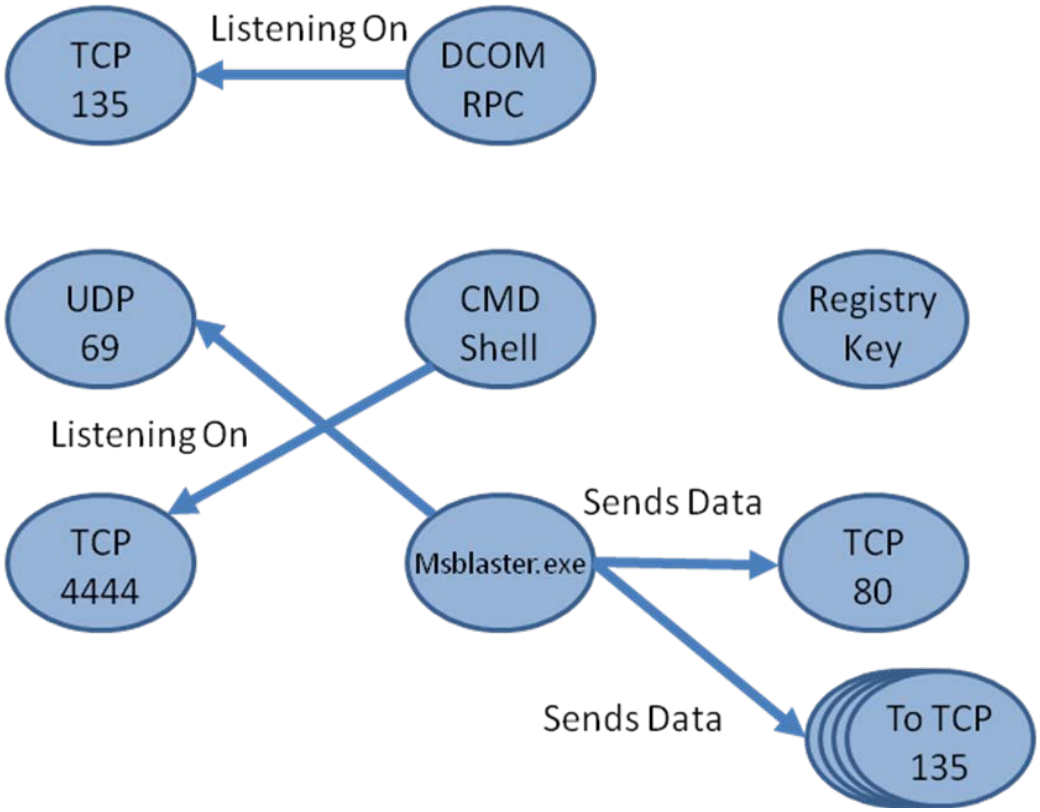

Figure 17. MSBlaster.exe attempting to propagate

The Blaster example demonstrates this model can be applied to malware and computer systems in general. It also illustrates the importance of combining information from across the multiple model layers to create a hypothesis of what is happening to the system. However, this example only focused on modeling and still needs to have the full prediction kernel applied to it. Further research and discussion will determine whether anything would prevent the kernel from being able to learn "Blaster-ness" and use that quale to recognize other worms.

5.4 Applications to the agent architecture

This model and kernel provides an implementation that can be used internally by individual agents to represent their world model in a manner that not only uses the relational aspects of qualia, but also takes time and event history into account. Each agent takes stimuli and stimuli context from the outside world and uses that data to generate entity-relationship representations based on current knowledge. The main process remains the same, and the only difference is how the model is represented.

Patterns provide a useful representation, as they are evoked by the changes in the model and not just the current state, providing a temporal aspect lacking from the initial agent system. Agents using a pattern representation would be able to track changes in the world over time and use that in their decision-making process. A pattern-based kernel provides prediction and error-correcting absent in the current agent architecture. Model graphs for each agent can be adjusted based on the context of the rest of the system, forming a plausible narrative and influencing the pattern predictions based on that plausible narrative. Also, multi-layer patterns can be created by using the hierarchical nature of the agent system. An evoked pattern in a child agent can be passed to parent agents as stimuli context which the parent agent can use as data to modify its representation of the world.

5.5 Summary

This chapter describes a novel method of representing the world and a kernel to process the resulting model which is applied Blaster worm to demonstrate its ability to model a computer system. The pattern representation and kernel offer an example of a qualia-based representation that can be used with the agent architecture developed in Chapter IV. The pattern representation and kernel and other qualia-based implementations offer alternative internal implementations for the individual agents in the system. Replacing traditional implementation such as multi-layer perceptrons with qualia-based representations will improve the agent architecture's functionality by allowing it to more efficiently share context and model changes over time.

VI Conclusion

6.1 Overview

This chapter discusses the contributions of this research and examines future work. It describes how the research met its goals and concludes with a summary of the research.

6.2 Contributions

The research makes several contributions malware detection and pattern recognition. It provides the definitions, framework, and design for a new decision-making architecture. Further, it develops a novel way of representing the world and how it changes over time. Each of these contributions are discussed in detail below.

6.2.1 Definitions of Stimuli, Data, Information, Knowledge, and Context

Stimuli, data, information, knowledge, and context are unique to the entity or agent processing them. Without providing a reference entity, it is impossible to discuss these terms in a meaningful fashion. This new definition provides a consistent terminology to discuss the new agent architecture.

6.2.2 Defines the Layers, Entities, and Relationships in the Cyber Domain

The definition of main layers, entities, and relationships of the cyber domain provide a consistent basis set for qualia-based models. A qualia-based model for the cyber domain must represent these layers with the proper entities and relationships associated with each layer.

6.2.3 Agent Architecture

The main contribution of this research is the development of the agent architecture which incorporates QUEST tenets. It provides a methodology for individual agents to update their representations based on their perception of the world. That perception is adjusted based on the results of other agents, forming a cohesive system-level model of the world. This is a unique

approach because it allows agents to accept context from other agents, meaning that their perception is no longer driven simply by their sensors, but also their relationships with the other agents. This subjective nature is consistent with QUEST, and the architecture is general enough that it can be applied to a large number of the QUEST driver problems.

6.2.4 Pattern Representation

In addition to the agent architecture, this research also described a representation that could be used internally by the agents to add a temporal component to their calculations. The representation uses a graph theory based model to define the world and uses past patterns of behavior to predict future events and explain the current state of the world. This representation models the Blaster Worm in operation.

6.3 Future Work

The architecture provides improvements over current techniques, but several issues need to be addressed such as convergence of the system, automated structure learning, and testing of an implementation. This section examines these issues and offers ideas for future solutions.

6.3.1 Convergence Issues

As discussed in Section 4.7.3, it is unknown whether repeating the narrative refinement process will result in system-wide convergence. While each iteration converges to a distinct solution, there is no guarantee that repeated iterations will converge to a plausible narrative that best represents the environment. Each iteration updates the representations of the agents in the system based on the representations of other agents which were affected by the last iteration. Because each iteration is dependent on the last one, it is possible that repeated iterations will actually degrade, rather than improve the system's perception of the world. The system's perception may change between multiple plausible narratives as iterations continue.

Future work will need to determine whether convergence is required to provide an engineering advantage and how it can be made to converge if needed. General convergence may not be required, as the system could function with a limited number of iterations until certain conditions. Further, if the system does converge to a single plausible narrative, that could be an indication of the system's certainty. If the system does not converge, polling the results over the iterations could provide a method of determining which plausible narratives are being evoked the most.

6.3.2 Algorithms for Determining Agent and System Structures/Connectivity

While briefly touched on in Section 4.9, the problem of automatically creating the system layout is still a concern. The system needs a way to automatically determine the number of agents, how the agents are connected, the internal representation of the agents, and what parts of the representations are shared as context. Further, the system needs to be able to add and remove agents as the process goes on. Manually defining them can be useful to a point, being able to perform this process automatically would greatly enhance the effectiveness of the system. One method that bears investigation is the use of LDA as described in Section 6.3.3.3.

6.3.3 Implementation of Agent Model

The architecture still requires testing to determine whether it provides an engineering advantage over current techniques. Malware detection is a suitable driver problem for this purpose because of its subjective nature and its malleability. If the system performs well at detecting malware, it is likely to be useful in other domain areas. The individual agent architecture needs to be examined first and then the system of agents can be tested.

6.3.3.1 Single Agent Proof of Concept

The first step in testing the proposed system is verifying the performance of a single agent as a baseline for performance of a multi-agent system. The output of a single agent is its internal representation and kernel implementation unless context is applied. A simple test implements a basic agent and provide simulated shared-representation context as if it was coming from another agent. A malware example could be tested using an SVM-based agent that takes LDA topics of a program as stimuli and generates a malicious or benign representation. The agent could also have a shared-representation context input that represents the current "threat level" of the system. The threat level context would drive the agent's perception of malicious or benign. Testing this implementation is straight-forward as there are a large number of data sets that label programs as malicious or benign.

6.3.3.2 Multi-layer, Multi-Agent proof of concept

To implement and test a multi-layer, multi-agent system, several parameters must be determined. First, the overall architecture of the system must be specified, including the number of agents, the number of levels and the initial connections. As described in Section 6.3.2, automatically defining the system is an open problem. Therefore, the overall architecture must be defined manually.

After the system-level architecture is defined, the internal representation and agent kernels are determined. For initial testing, the same representation and kernel should be used for all of the agents to simplify the shared-representation context transfers. While agent knowledge can translate disparate representations between agents, keeping the representations in the same format makes it easier to test and still allows for the sharing of context. Using multi-layer

perceptrons (MLP) for the individual agents allows the Libetian performance of the system to be compared to a monolithic MLP with the same structure and similar stimuli paths.

6.3.3.3 LDA proof of concept

Even with a defined architecture, agents still rely on labeled data for training. Depending on the domain area, the needed data may not be available. For example, in the malware domain, most datasets simply have labels of malicious or benign. Between the opcodes and low-level metadata, there are few if any intermediate labels useful for training individual agents. Artificially generated labels through the use of LDA to derive topics are a possibility.

In Section 3.4.3 LDA was used to generate 100 topics that served as abstractions of functions from opcode sequences. These topics are the labels for the training data for a system of agents that reproduces that experiment using the proposed architecture. The system would consist of 101 individual agents arranged in two layers. The bottom layer of 100 agents would accept opcode sequences as stimuli and provide an evocation of one of the topics as a result. These results would be sent to the lone parent agent as stimuli context. This agent would determine whether programs were malicious or benign and pass that result to the child agents as shared-representation context. Each of the bottom agents would be trained using the LDA topic labels while the upper agent would be trained on the malicious/benign labels. Additionally, the actual LDA algorithm could serve as another agent at the bottom layer, providing shared-representation context to all of the topic agents. The system could then be compared to the original experiment to determine whether it has better performance.

6.3.4 Temporal Representation

The current architecture does not explicitly take past events into account when determining the current state or predict future states. To make the system more qualia-based, it

must take the temporal aspect of the world into account. Changes in the world are just as important to correct decision-making as the current state. This functionality can be either added to the architecture at the system-level or by changing the internal model and kernel of the agents. Modifying the system-wide architecture requires an additional temporal kernel to track changes in the world and update the representations of the agents based on the temporal events. Changing the internal implementation of the individual agents is a better option, as it would have no effect on the overall operation of the system. Each agent would simply use an internal model representation and processing kernel that incorporates time rather than the basic MLP solutions described in Section 4.5. Possible solutions include Hidden Markov Models, Recurrent Neural Networks, and the Pattern Representation described in Section 5.2. All of these are capable of modeling temporal changes and can be used by agents to represent the world.

6.3.4.1 Further Development and Implementation of the Pattern Representation

The pattern model and kernel described in Section 5.2 are conceptual and have not been implemented or tested. They provide a unique representation of the world and are worth considering for future development, implementation, and testing. Implementing this representation provides an additional tool for the agent architecture as they account for changes in the world and use them to predict future events.

6.4 Research Goals

The success of the research is defined by how well it met its goals. The main goal of this research is to apply QUEST tenets to the malware detection domain to achieve an engineering advantage over current techniques. The research succeeds in applying the tenets to malware detection and provides a resulting agent architecture and representation.

6.4.1 Examine Current Malware Techniques

The research examined the current malware detection methods and discussed their limitations. Current methods focus too much on individual pieces of low-level, implementation specific data and are unable to generate a "big picture" view of the entire system. While low-level approaches are useful for known strains of malware, they are limited in their ability to detect new strains because they do not understand fundamentally what makes a piece of software malicious.

6.4.2 Define a General Model for the Cyber Domain

The research successfully outlined a multi-layer model of the cyber domain, stretching from the physical layer to the network level. The model incorporates cyber entities and relationships, providing a qualia-based model for use in future designs and development. The model used the Blaster worm to show it is capable of representing malware.

6.4.3 Develop a Qualia-based Architecture

The most important goal of the research was to create a general, qualia-based architecture that can not only be applied to malware detection but also to other difficult pattern recognition problems. The research generates such an architecture through the use of multiple levels of individual agents which share context to generate a system-level plausible narrative of the events in the world. The architecture offers the potential to manipulate and process sensor stimuli and data subjectively, making it well-suited for malware detection and other domains where context plays an important role. The architecture is general enough that it can be applied to other domains and several potential representations and kernels that can be used to implement the agents are discussed. One of these representations is a novel pattern-based model developed specifically for this research.

6.4.4 Discuss Possible Pitfalls and Future Enhancements

The final portion of the research examines the next steps for future work. It describes the issues that an implementation must overcome, including several ways that they can be mitigated. It also outlines the steps needed to implement and test single agents and multi-layer, multi-agent architectures. It also describes how to generate labeled test data for these implementation tests using the LDA algorithm

6.5 Summary

This research explored novel ways to detect malware through the use of QUEST and qualia-based solutions. These solutions focus on higher-level concepts and abstractions of the system, allowing them to detect malicious software, regardless of the lowest-level implementations. The research developed a general agent architecture that performs pattern recognition in a qualia-based manner and is scalable and general enough to apply to any number of other QUEST driver problems. Future testing will determine whether this architecture provides a general solution to malware detection that can keep up with the constantly changing and expanding threat of malicious software.

Bibliography

[1] M. Hines, "Malware flood driving new AV," in *InfoWorld*, 2007.

[2] P. Szor, *The art of computer virus research and defense*. Upper Saddle River, NJ: Addison-Wesley, 2005.

[3] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith, "Malware normalization," University of Wisconsin, Madison, 2005.

[4] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia, "Normalizing Metamorphic Malware Using Term Rewriting," in *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*: IEEE Computer Society, 2006.

[5] D. Bruschi, L. Martignoni, and M. Monga, "Using code normalization for fighting self-mutating malware," in *Conference on Detection of Intrusions and Malware and Vulnerability Assessment*: IEEE Computer Society, 2006.

[6] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-Aware Malware Detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*: IEEE Computer Society, 2005.

[7] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data Mining Methods for Detection of New Malicious Executables," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, 2001.

[8] J. Z. Kolter and M. A. Maloof, "Learning to Detect and Classify Malicious Executables in the Wild," *The Journal of Machine Learning Research* vol. 7, pp. 2721-2744, 2006.

[9] C. Krugel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic Worm Detection Using Structural Information of Executables," in *RAID*, 2005.

[10] M. Christodorescu, S. Jha, and C. Kruegel, "Mining Specifications of Malicious Behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Dubrovnik, Croatia: ACM, 2007.

[11] W. Li, L.-c. Lam, and T.-c. Chiueh, "Accurate Application-Specific Sandboxing for Win32/Intel Binaries," in *Proceedings of the Third International Symposium on Information Assurance and Security*: IEEE Computer Society, 2007.

[12] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham, "Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code," in *Proceedings of the 2003 ACM workshop on Rapid malcode*. Washington, DC, USA: ACM, 2003.

[13] A. G. Tokhtabayev and V. A. Skormin, "Non-Stationary Markov Models and Anomaly Propagation Analysis in IDS," in *Proceedings of the Third International Symposium on Information Assurance and Security*: IEEE Computer Society, 2007.

[14] H. Qiao, J. Peng, C. Feng, and J. W. Rozenblit, "Behavior Analysis-Based Learning Framework for Host Level Intrusion Detection," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*: IEEE Computer Society, 2007.

[15] A. Shabtai, M. Atlas, Y. Shahar, and Y. Elovici, "Evaluation of a Temporal-Abstraction Knowledge Acquisition Tool in the Network Security Domain," in *Proceedings of the 4th International Conference on Knowledge Capture*. Whistler, BC, Canada: ACM, 2007.

[16] H. Aytug, F. Boylu, and G. J. Koehler, "Learning in the Presence of Self-Interested Agents," in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences - Volume 07*: IEEE Computer Society, 2006.
[17] S. K. Rogers, M. Kabrisky, K. Bauer, and M. Oxley, "Computing Machinery and Intelligence Amplification," in *Computational Intelligence, The Experts Speak (Chapter 3)*. New Jersey: IEEE Press, 2003.
[18] S. K. Rogers, C. Sadowski, K. W. Bauer, M. E. Oxley, M. Kabrisky, A. Rogers, and S. D. Mott, "The life and death of ATR/sensor fusion and the hope for resurrection," in *Automatic Target Recognition XVIII*, vol. 6967, A. S. Firooz and M. Abhijit, Eds. Orlando, FL: Proceedings of the SPIE, 2008, pp. 696-702.
[19] M. Karresand, *A Proposed Taxonomy of Software Weapons*. Linköping, Sweden: Linköping University, 2003.
[20] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993-1022, 2003.
[21] X. Wang, X. Ma, and W. E. L. Grimson, "Unsupervised Activity Perception in Crowded and Complicated Scenes Using Hierarchical Bayesian Models," *IEEE Transactions on Pattern Analysis and Machine Intelligence* vol. 31, pp. 539-555, 2009.
[22] Wikipedia.org, "Von Neumann Architecture." http://en.wikipedia.org/wiki/Von_Neumann_architecture, 2006.
[23] Wikipedia.org, "Image:Von Neumann architecture.svg." http://en.wikipedia.org/wiki/Image:Von_Neumann_architecture.svg, 2006.
[24] Microsoft, "Windows registry information for advanced users." http://support.microsoft.com/kb/256986, 2008.
[25] B. Birrer, R. Raines, R. Baldwin, M. Oxley, and S. Rogers, "Using Qualia and Multi-Layered Relationships in Malware Detection " in *IEEE Symposium Series on Computational Intelligence in Cyber Security*. Nashville, TN: IEEE Press, 2009.
[26] B. D. Birrer, R. A. Raines, R. O. Baldwin, M. E. Oxley, and S. K. Rogers, "Using qualia and novel representations in malware detection," 2009.
[27] "VX Heavens." http://vx.netlux.org/.
[28] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici, "Unknown Malcode Detection Using OPCODE Representation," in *Proceedings of the 1st European Conference on Intelligence and Security Informatics*. Esbjerg, Denmark: Springer-Verlag, 2008.
[29] "InstructionCounter plugin for IDA Pro." http://www.the-interweb.com/serendipity/index.php?/archives/57-InstructionCounter-plugin-for-IDA-Pro.html, 2005.
[30] B. D. Birrer, R. A. Raines, R. O. Baldwin, M. E. Oxley, and S. K. Rogers, "Using Qualia and Hierarchical Models in Malware Detection," *Journal of Information Assurance and Security* vol. 4, pp. 247-255, 2009.
[31] B. Libet, "Do we have free will?," *Journal of Consciousness Studies*, vol. 6, pp. 47-57, 1999.
[32] L. K. Komatsu, "Recent views of conceptual structure," *Psychological Bulletin*, vol. 112, pp. 500-526, 1992.
[33] M. Garey and D. Johnson, *Computers and Intractability - A Guide to the Theory of NP-completeness*. New York: W.H. Freeman, 1979.

[34] M. Bailey, E. Cooke, F. Jahanian, D. Watson, and J. Nazario, "The Blaster Worm: Then and Now," in *IEEE Security and Privacy*, vol. 3, July/Aug 2005 ed, 2005, pp. 26-31.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
25-03-2010	Doctoral Dissertation	March 2007-March 2010

4. TITLE AND SUBTITLE
DEVELOPING A QUALIA-BASED MULTI-AGENT ARCHITECTURE FOR USE IN MALWARE DETECTION

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Birrer, Bobby, D.

5d. PROJECT NUMBER
N/A

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
N/A

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management
2950 Hobson Way, Building 640
Wright Patterson AFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/DCS/ENG/10-01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Steven K. Rogers
Air Force Research Laboratory, Sensors Directorate
13th St, Bldg 620
Wright Patterson AFB OH 45433
DSN: 674-9891 Email: Steven.Rogers@WPAFB.AF.MIL

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RY

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Detecting network intruders and malicious software is a significant problem for network administrators and security experts. New threats are emerging at an increasing rate, and current signature and statistics-based techniques are not keeping pace. Intelligent systems that can adapt to new threats are needed to mitigate these new strains of malware as they are released. This research detects malware based on its qualia, or essence rather than its low-level implementation details. By looking for the underlying concepts that make a piece of software malicious, this research avoids the pitfalls of static solutions that focus on predefined bit sequence signatures or anomaly thresholds.

This research develops a novel, hierarchical modeling method to represent a computing system and demonstrates the representation's effectiveness by modeling the Blaster worm. Using Latent Dirichlet Allocation and Support Vector Machines abstract concepts are automatically generated that can be used in the hierarchical model for malware detection. Finally, the research outlines a novel system that uses multiple levels of individual software agents that sharing contextual relationships and information across different levels of abstraction to make decisions. This qualia-based system provides a framework for developing intelligent classification and decision-making systems for a number of application areas.

15. SUBJECT TERMS
Anti-virus, Malware Detection, Qualia, Agents

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
U	U	U	UU	112	(937) 255-6565, ext 4278 Email: Richard.Raines@afit.edu

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18